University of Alberta

FILE MEMORY FOR EXTENDED STORAGE DISK CACHES

by

John C. Koob ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Spring 2004

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-96502-3
Our file Notre référence
ISBN: 0-612-96502-3

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

This thesis investigates the application of semiconductor file memory as extended storage. While magnetic disk technology has shown remarkable improvements in capacity and cost, a widening access time gap has developed between disk and main memory. Instead of increasing main memory capacity, experimental file memory could fill the access time gap since it is slower, but more economical per bit, than conventional random access memory. With suitable fault tolerance, file memory would function as a disk cache in a stage of the memory hierarchy known as extended storage. While common in legacy mainframes, a distinct hierarchy stage between main memory and disk is absent in modern systems. This research argues for the reintroduction of extended storage to improve performance while remaining cost-effective. Emulated using a modified Linux 2.4.18 operating system kernel, file memory is shown to reduce system costs by replacing the 27% smaller virtual disk cache in main memory and achieve equivalent performance.

We are . . . forced to recognize the possibility of
constructing a hierarchy of memories, each of which
has a greater capacity than the preceding but which
is less quickly accessible.

— A. W. Burks, H. H. Goldstine, and J. von Neumann, 1946

To Mom and Dad

Acknowledgements

This research was funded by scholarships from the Natural Sciences and Engineering Research Council of Canada (NSERC), the Alberta Informatics Circle of Research Excellence (iCORE), and the University of Alberta. Additional funding and support for research equipment was provided by Micronet R&D, MOSAID Technologies Inc., ATMOS Corporation, and the Canadian Microelectronics Corporation.

I must thank my supervisor, Dr. Duncan Elliott, for providing opportunities for in-depth research on a variety of topics and for his suggestions throughout my thesis research. Whenever I had to prepare a paper for publication, I could rely on Dr. Bruce Cockburn to provide helpful recommendations. I wish to thank my colleagues for their assistance and feedback: Dan Leder, Craig Joly, Tyler Brandon, Steve Dillen, Raymond Sung, Kris Breen, and Christian Giasson. Finally, I must thank Sue Ann Ung for all of her enthusiastic help and encouragement.

Table of Contents

1 Introduction **1**

1.1 Introduction . . . 1
1.2 Motivation . . . 1
1.3 Overview . . . 3
  1.3.1 Concepts . . . 3
  1.3.2 Fault Tolerance . . . 6
  1.3.3 ESDC Design . . . 7
1.4 Applications of Extended Storage . . . 8
  1.4.1 Personal Computer Systems . . . 8
  1.4.2 Portable Devices . . . 8
  1.4.3 Downgraded Memory . . . 9
  1.4.4 MEMS Storage Technology . . . 9
  1.4.5 Multilevel DRAM . . . 10
1.5 Conclusion . . . 10

2 Concepts **11**
2.1 Introduction . . . 11
2.2 File Memory as Extended Storage . . . 12
  2.2.1 The Performance Gap . . . 12
  2.2.2 Economical File Memory . . . 13
2.3 Extended Storage Architectures . . . 15
  2.3.1 Extended Storage Hierarchy . . . 15
  2.3.2 Disadvantages of Extended Storage Hierarchies . . . 19
  2.3.3 Historical Study: The IBM 3090 Mainframe . . . 20
  2.3.4 Historical Study: The Cray Y-MP Mainframe . . . 21
  2.3.5 Summary . . . 22
2.4 Extended Storage Disk Architectures . . . 22
  2.4.1 Disk Caching Disk . . . 23
  2.4.2 Tertiary Storage . . . 24
2.5 Architectural Support for Extended Storage . . . 24
  2.5.1 Compressed Caching . . . 25
  2.5.2 Log-Structured File Systems . . . 26
  2.5.3 Conquest File System . . . 26

2.6 Reliability versus Performance 27
  2.6.1 Partially-Safe Disk Caches 28
  2.6.2 The Rio File Cache 29
2.7 Conclusion 30

3 Fault Tolerance **31**
3.1 Introduction 31
3.2 Methods of Fault Tolerance 32
  3.2.1 Fault Tolerance for Disks 32
  3.2.2 Fault Tolerance for Caches 33
  3.2.3 Fault Tolerance for Flash 34
  3.2.4 Wafer-Scale File Memory 35
  3.2.5 Error-Correcting Codes and Redundancy 36
3.3 Device-Level Bad Block Marking 43
  3.3.1 Partially-Good Product 43
  3.3.2 Methods and Results 45
  3.3.3 Discussion 46
3.4 System-Level Bad Block Marking 46
  3.4.1 Advantages 47
  3.4.2 Binary Buddy System for Memory Allocation 47
  3.4.3 Binary Buddy System for Fault-Tolerance 50
  3.4.4 Feasibility 54
3.5 Conclusion 57

4 Extended Storage Disk Cache **59**
4.1 Introduction 59
4.2 ESDC Design Overview 59
4.3 ESDC Design Principles 62
4.4 High Memory Management 63
  4.4.1 Memory Address Space 63
  4.4.2 Kernel Address Space 64
  4.4.3 Memory Zones 65
  4.4.4 High Memory Emulation 66
4.5 Memory Hierarchy Integration 67
  4.5.1 Hierarchy Properties 69
  4.5.2 Pages Excluded from ESDC 70
4.6 ESDC Page Containment 72
  4.6.1 Architectural Considerations 73
  4.6.2 Bounce Buffers 75
  4.6.3 Page Allocation 76
4.7 Configurable Performance 77
  4.7.1 Design Alternatives 77
  4.7.2 Implementation Issues 78
4.8 Caching Properties of ESDC 79

4.9 Demand Paging with ESDC 82
  4.9.1 Demand Paging 84
  4.9.2 Copy On Write 86
4.10 Metrics Acquisition 86
  4.10.1 ESDC and the `proc` File System 86
  4.10.2 ESDC Access Statistics 88
4.11 Implementation Robustness 89
4.12 Conclusion 90

5 Experiments **93**
5.1 Introduction 93
5.2 Experimental Methodology 94
  5.2.1 Cost-Effectiveness Evaluation 94
  5.2.2 Experimental Platform 95
  5.2.3 Experimental Automation 95
  5.2.4 Alternative Models 97
5.3 Experimental Validation 98
  5.3.1 Sources of Experimental Error 98
  5.3.2 ESDC Metrics Verification 99
5.4 ESDC Experiments 101
  5.4.1 Experimental Suite 101
  5.4.2 PostMark Synthetic Benchmark Results 102
  5.4.3 Bonnie Synthetic Benchmark Results 114
  5.4.4 MUMmer Application Workload Results 121
  5.4.5 Kernel Compilation Workload Results 124
5.5 Conclusion 127

6 Conclusion **129**
6.1 Introduction 129
6.2 Design Summary 129
  6.2.1 Fundamentals 129
  6.2.2 Limitations 130
6.3 Summary of Results 131
  6.3.1 PostMark Benchmark 131
  6.3.2 Bonnie Benchmark 132
  6.3.3 MUMmer Application 133
  6.3.4 Kernel Compilation Application 134
6.4 Conclusion 134

Bibliography **135**

A Implementation **143**
A.1 Selected Kernel Modifications for ESDC 143
A.1.1 High Memory Emulation 143
A.1.2 Utilizing GFP Flags for ESDC 145
A.1.3 Configurable Performance Implementation 148
A.1.4 ESDC Access Statistics 151

B Operating System Modifications **153**
B.1 ESDC Kernel Patch 153

C Experimental Automation Scripts **173**
C.1 Remote Experimental Platform Support 173
C.2 Automation and Data Acquisition 177
C.3 Experimental Data Visualization 196

List of Tables

2.1 Memory Hierarchy Characteristics 12
2.2 Examples of Extended Storage Implementations 20
2.3 Impact of Disk Cache Write Policy 27

3.1 Bad Block Data for Toshiba's NAND Flash Memories 34
3.2 IBM 16-Mb DRAM and Samsung 1-Gb DRAM 38
3.3 Yields for DRAMs with Different Redundancy Schemes 39
3.4 Estimate of Bad Block Marking Resources 57

4.1 High Memory Configuration Parameters 67
4.2 Page Cache and Buffer Cache 71
4.3 ESDC Metrics 87

5.1 Experimental Platform 95
5.2 ESDC Read Test 99
5.3 ESDC Read and Write Test 100
5.4 Experimental Suite 101
5.5 Impact of ESDC with File Memory Access Time Ratio of 2 109
5.6 Analysis of Behavior Caused by Kernel Race Condition 117
5.7 Results of the MUMmer Experiments 122
5.8 Kernel Compilation Measurements for Uncached ESDC 125

A.1 `__GFP_HIGHMEM` Analysis 146
A.2 `__GFP_HIGHUSER` Analysis 147

List of Figures

1.1 Various technologies used in a memory hierarchy 4
1.2 A comparison of SRAM and DRAM cells 5

2.1 Extended storage locations in the memory hierarchy 16
2.2 Solid-state disk in a memory hierarchy 18
2.3 The IBM 3090 and Cray Y-MP memory hierarchies 21
2.4 The disk caching disk (DCD) hierarchy 23
2.5 Non-volatile RAM cache models 28

3.1 Wafer-scale RAM with fault tolerance 35
3.2 Block diagram of 16-Mb DRAM 37
3.3 Yields for the IBM 16-Mb DRAM 40
3.4 Yields for the Samsung 1-Gb DRAM 41
3.5 Buddy system data structures 49
3.6 Bad block marking by block allocation feigning 51
3.7 Downgraded DRAM classification process 55

4.1 ESDC memory hierarchy 60
4.2 Virtual address space in Linux 63
4.3 Kernel address space 64
4.4 Zoning of physical memory 65
4.5 Original page cache architecture with page dispersion 72
4.6 ESDC architecture featuring page containment 73
4.7 Algorithm to create aligned MTRR ranges within ESDC 81
4.8 Simplified overview of page fault exception handling in Linux . . . 83

5.1 PostMark performance using cached DRAM 103
5.2 PostMark read rate comparing file memory with DRAM 105
5.3 PostMark write rate comparing file memory with DRAM 106
5.4 PostMark read performance for constant main memory size 107
5.5 PostMark write performance for constant main memory size 107
5.6 PostMark performance and file memory access times 110
5.7 PostMark performance and access times with 64-MB ESDC 111
5.8 PostMark performance and access times with 112-MB ESDC 111
5.9 ESDC miss rate for PostMark with 16-MB ESDC 113

5.10 ESDC miss rate for PostMark with 128-MB ESDC 113
5.11 Visualization of kernel race condition 116
5.12 Bonnie random seek rate comparing file memory with DRAM . . . 119
5.13 Bonnie random seek rate for constant main memory size 120
5.14 Kernel compilation execution time with cached ESDC 124
5.15 ESDC miss rate for kernel compilation with 128-MB ESDC 126

A.1 Function callgraph of `generic_file_write()` 148
A.2 Function callgraph of `generic_file_read()` 149

Nomenclature

List of Acronyms

COW	Copy on write, page 86
DCD	Disk caching disk, page 23
DMA	Direct memory access, page 66
DRAM	Dynamic random-access memory, page 4
ECC	Error-correcting code, page 14
ESDC	Extended storage disk cache, page 17
ISA	Industry standard architecture, page 66
LFS	Log-structured file system, page 26
LRU	Least-recently-used, page 68
MEMS	Micro-electromechanical systems, page 9
MTRR	Memory type range register, page 80
MUM	Maximal unique matches, page 121
NUMA	Non-uniform memory access, page 130
NVRAM	Non-volatile random-access memory, page 28
PCI	Peripheral component interconnect, page 79
PDA	Portable digital assistant, page 8
SEC-DED	Single-error correction/double-error detection, page 37
SIMM	Single in-line memory module, page 55
SRAM	Static random-access memory, page 3
SSD	Solid-state disk, page 17

List of Symbols

α	Defect clustering parameter, page 44
λ_K	Average number of killer defects, page 44
λ_{sc}	Average number of faults per cell, page 39
$addr_b$	Block address containing a fault, page 51
$addr_f$	Address of a faulty word, page 51
$addr_{kernel}$	Kernel address space size, page 65
$addr_{reserve}$	Address space reserved by the kernel, page 65
b_b	Number of bits in a book, page 40
b_c	Number of bits column, page 39
b_c	Number of bits in a column, page 43
b_r	Number of bits in a row, page 40
b_{cw}	Number of bits in a code word, page 41
b_{DRAM}	Number of bits in a DRAM, page 39
k	Order of a buddy system allocation, page 53
k_{max}	Maximum order of the buddy system, page 53
n_b	Number of books in a DRAM, page 41
n_c	Number of columns in a book, page 39
n_r	Number of rows in a section, page 40
n_s	Number of sections in a DRAM, page 40
n_{cw}	Number of code words in a book, page 41
r_c	Number of redundant columns per book, page 39
r_r	Number of redundant rows per section, page 40
s_k	Size of a buddy system bitmap, page 53
s_m	Size of memory managed by buddy system, page 53
s_p	Size of a page frame, page 53

s_{set}	Size of a set of buddy system bitmaps, page 53
Y_0	Gross yield of DRAM, page 44
Y_b	Yield of a given block of a DRAM, page 44
Y_K	Probability of no killer defects in a DRAM, page 44
Y_{AG}	All good yield of DRAM, page 44
Y_{book_E}	Yield of a book in a DRAM with ECC, page 41
$Y_{book_{RCE}}$	Effective yield of a book of DRAM with row and column redundancy plus ECC, page 43
$Y_{book_{RC}}$	Yield of a book of a DRAM with row and column redundancy, page 39
$Y_{col_{RCE}}$	Effective yield of a column of DRAM with row and column redundancy plus ECC, page 43
$Y_{col_{RC}}$	Yield of a column of DRAM, page 39
Y_{cw}	Yield of an ECC code word, page 41
Y_{DRAM_E}	Yield of a DRAM with ECC, page 41
$Y_{DRAM_{RCE}}$	Effective yield of a DRAM with row and column redundancy plus ECC, page 43
$Y_{DRAM_{RC}}$	Yield of a DRAM with row & column redundancy, page 40
Y_{DRAM}	Yield of a DRAM device, page 39
Y_{EQ}	Equivalent yield partially-good product, page 44
$Y_{row_{RCE}}$	Effective yield of a row of DRAM with row and column redundancy plus ECC, page 43
$Y_{row_{RC}}$	Yield of a row of DRAM with column redundancy, page 40
Y_{sc}	Yield of a single DRAM cell, page 39
$Y_{sec_{RCE}}$	Effective yield of a section of DRAM with row and column redundancy plus ECC, page 43
$Y_{sec_{RC}}$	Yield of a section of DRAM with row and column redundancy, page 40
$zone_{max}$	Highest kernel-addressable page frame address, page 65

List of Terms

anonymous page	A page not mapped to a file on disk, page 84
application workload	An application that attempts to emulate the behavior of a set of programs, page 101
asynchronous I/O	The I/O is initiated but the current process is not blocked until the I/O operation completes, page 28
backing store	The stage below a given stage of a hierarchy, but usually refers to the disk media that backs pages of main memory, page 85
bad block marking	A method of identifying a block of memory cells that contains one or more defects so that the block can be avoided or bypassed, page 14
book	The area of memory to which column redundancy is applied, page 37
bounce buffer	An intermediary required for performing I/O operations for data located in high memory due to addressing limitations of some I/O devices, page 75
buddy system	A fast dynamic memory allocation algorithm that manages blocks of powers of two in size and minimizes external fragmentation, page 47
buffer	A area of memory used to store a block of data associated with a block device, page 71
buffer page	A 4-KB page used for the allocation of one to eight buffers in memory, page 71
cache pollution	Occurs when another process or CPU replaces the useful lines of a cache that will be used soon, page 19
code word	A set of data bits and ECC check bits used for fault tolerance in memories, page 37
controller cache	A cache in a multi-disk controller that buffers data for command queuing, seek ordering, DMA transfers and other related functions, page 16
copy on write	Pages are shared between parent and child processes and only are copied upon writes; improves the performance of child process creation, page 86

demand paging	Virtual memory mechanism where pages are retrieved from backing store in the event of a page fault, page 84
device cache	A relatively small memory embedded into a disk drive or similar device to buffer data in a manner suitable to the mechanical requirements of the media, page 15
disk cache	Semiconductor memory that buffers explicit I/O operations or implicit I/O due to paging, page 11
downgraded DRAM	Inexpensive DRAM chips that failed the manufacturer's final testing stage but still have functional areas of memory, page 9
error-correcting code	The use of check bits stored with a set of data bits that can be used to correct memory errors, page 14
expanded storage	Introduced on the IBM 3090, a form of extended storage that is accessed via synchronous transfers of 4-KB pages between itself and main memory, page 20
experiment	A synthetic benchmark or an application workload used to empirically evaluate ESDC, page 101
ext2	A file system developed for the Linux operating system supporting up to four terabytes, page 71
extended storage	Page-addressable memory that is used for purposes other than general-purpose main memory, page 16
feigned allocation	An allocation of page frames for the purpose of marking bad blocks rather than for normal use as storage for pages, page 51
file memory	A type of economical memory suitable for use as *extended storage*. It describes block-addressable memory that uses error correction, redundancy and bad block marking to reduce the average cost per working bit [82], page 5
fully-associative	Cache placement policy where a line can be placed anywhere in the cache, page 69
industry standard architecture	A legacy bus architecture that still affects operating system memory management, page 66
kernel address space	The portion of the virtual address space exclusively mapped to the kernel, but the user address space also is accessible by the kernel, page 64

kernel mode	The execution state where the kernel is able to perform an operating system service, page 63
least-recently-used	A cache line replacement policy that is also employed in memory management architectures, page 68
log-structured file system	A file system that speeds up disk writes, assuming that disk caches are dominated by read requests, page 26
major fault	A page fault where a page must be retrieved from backing store; an expensive operation that blocks the execution of the current process, page 83
memory hierarchy	A hierarchy with small, slow, and expensive memory levels placed above larger and faster memories, page 3
memory region	Non-overlapping areas of memory used to describe virtual memory areas, page 84
memory type range register	A set of registers in the Intel architecture to control the types of caching for specific regions of memory, page 80
memory-mapping	A memory region associated with a portion of a file or device, page 84
migration	The process in MVS/ESA where pages are moved from expanded storage through central storage (main memory) to auxiliary storage (disk), page 21
minor fault	A page fault where a page can be located or allocated in memory without accessing backing store and blocking the current process, page 83
page cache	A disk cache of pages that are backed by files or devices; allocated in unused areas of memory, page 67
page descriptor	A data structure that stores information related to a page frame, page 48
page frame	A division of main memory that contains a page, which is usually 4 KB, page 48
peripheral component interconnect	A modern bus architecture for connecting peripherals in personal computers and workstations, page 79
proc file system	A virtual file system interface used to monitor or adjust kernel parameters, page 86

section	The area of memory to which row redundancy is applied, page 37
self-caching application	An application that wishes to bypass disk caches managed by an operating system due to unique performance considerations, page 72
solid-state disk	Devices that use non-volatile RAM as storage media instead of slower magnetic media, page 17
swap	Backing store for demand paging in Linux, which usually is a disk partition, page 83
swap cache	A subset of the page cache whose pages are replicated to one or more swap areas, page 85
swapped page	Technically, swapping involves moving the virtual memory pages of an entire process to backing store to free physical memory. Linux kernel documentation and this thesis refer to the individual virtual memory pages that have been paged out to backing store as swapped pages, page 83
synchronous I/O	The current process is blocked until the I/O operation completes, page 28
synthetic benchmark	An artificial algorithm that attempts to emulate the behavior of a set of programs, page 101
user address space	The flat linear address space available to a user process running in user mode, page 64
user mode	An execution state where the process can not directly access kernel data structures, page 63
virtual disk cache	A page-addressable operating system disk cache that caches I/O operations using a variable number of free pages. It is also known as a main memory disk cache or as a page cache in Linux, page 17

Chapter 1

Introduction

1.1 Introduction

This thesis proposes that file memory should be incorporated as extended storage in the memory hierarchy of modern computer systems. Promoting a new memory architecture will raise questions of cost, capacity and performance. The purpose of this work is to determine the performance improvements provided by various extended storage configurations to quantify the costs of potential file memory products.

1.2 Motivation

Conventional digital systems use magnetic or optical disks for storage due to the need for high capacities at low cost. Disk technology has improved rapidly during the past decades, but the improvements have focused on cost and capacity instead of performance. The rate of improvement of transfer bandwidth and access time is lower than the increases in processor speed over time [56]. Disk arrays can improve the transfer bandwidth, but it generally is difficult to improve the access time for a mechanical device [56]. Since processor power doubles in speed each year while raw seek time only improves linearly, a widening gap has developed between processor performance and disk speeds [49]. A similar performance gap also is present between the processor and main memory. Semiconductor memory density doubles every two years, but the memory access time improvement of 7%

per year is much lower than the 55% annual increase in processor performance [25, p. 391]. Extensive research in cache design has focused on the processor-memory performance gap [25, p. 509]. However, the more substantial performance gap between memory and disk has received much less attention since I/O typically is buffered by using larger memories or solid-state disks [49]. Therefore, alternative technologies should be examined for reducing the large access time gap between memory and disk in a memory hierarchy.

When compared to solid-state random access memory, rotating disk media has relatively low ruggedness and orders of magnitude higher latency for random accesses. Semiconductor file memories offer an alternative to conventional disk storage by providing high capacities with the benefits of high reliability and low latency. Since ordinary semiconductor memory is very expensive relative to disk media of the same capacity, file memory must be made significantly more economical. By employing dynamic random access memory (DRAM) products with less than 100% nominal capacity, economical file memory could become a reality.

Recent research has shown that file memory could be fabricated more economically than conventional dynamic memory if a number of standard requirements of random access memory are relaxed [81, 82]. First, blocks or sectors of file memory are accessed instead of providing the capability for random accesses within a contiguous address space. Second, block access permits the use of error-correction methods to hide defective memory cells and increase manufacturing yield. That is, bad block management techniques similar to those used for disk or flash technology could reduce the number of rejected memory chips produced during manufacturing. Third, additional cost savings in memory fabrication can come from allowing more bits to be stored in the same silicon area by using multiple bits per cell [78]. While semiconductor disks have been proposed previously, the performance of a system equipped with file memory is an unresolved issue.

File memory, not only suitable for file storage applications, could be introduced into the memory hierarchy of a computer system to improve overall I/O performance. Since file memory is more economical than conventional DRAM, it conceptually resides between main memory and disk in terms of both cost and performance. This level of the memory hierarchy, known as extended storage in legacy mainframes, features slow, inexpensive memory that improves system performance by reducing I/O latencies. Currently, the challenge of reducing access latency for disk media is addressed by a virtual disk cache dynamically allocated in main memory. However, existing virtual disk caches that use a portion of main memory are several orders of magnitude smaller than disks due to the large disparity in cost between the two types of media. Due to these size restrictions, these caches are unable to fully exploit the locality of data during I/O transfers, especially with the large working set sizes prevalent in modern systems [28]. Therefore, the access time gap can be reduced further if large and inexpensive file memories are exploited as a disk cache. Thus, this work proposes a design for a new and distinct memory hierarchy level, an extended storage disk cache (ESDC). To quantify the cost-effectiveness of file memory as ESDC, a Linux operating system has been modified so that the overall performance can be evaluated with an experimental suite.

1.3 Overview

1.3.1 Concepts

Unfortunately, a memory technology with the best performance is also the most expensive. Fast access to large memory sizes is achieved using a *memory hierarchy*, which places small, expensive memories above slower, high-capacity memories (see Figure 1.1). Hierarchies of caches are popular between the processor and main memory, but lower levels of storage can also be designed as a hierarchy. In particular, hierarchy levels known as extended storage have appeared between main memory and disk in a number of architectures [4, 49, 50, 77].

1.3.1.1 Memory Hierarchy and Technology

The highest levels of a memory hierarchy commonly use fast but expensive static random-access memories (SRAMs). A typical CMOS SRAM cell consists of six

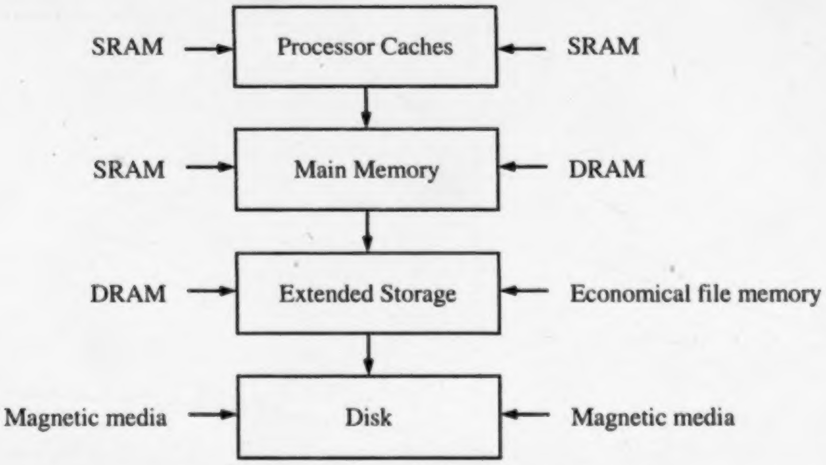
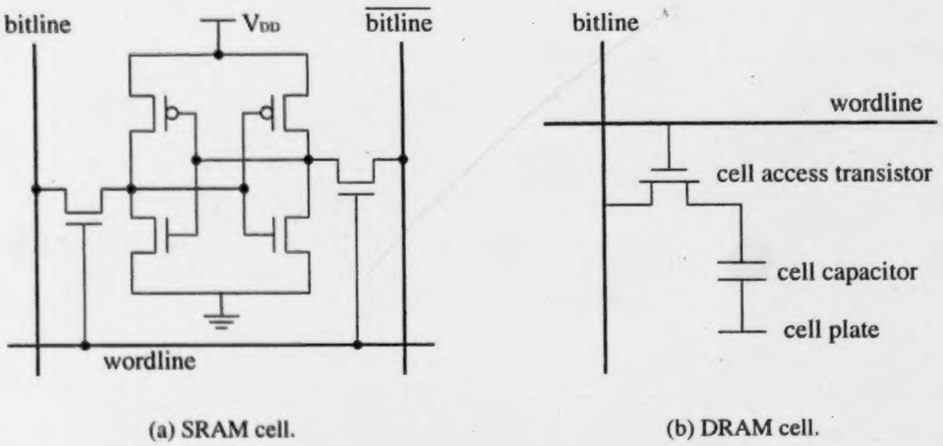


Figure 1.1: Supercomputers had an expensive hierarchy (left), while modern systems could benefit from economical file memory (right).

transistors (Figure 1.2(a)). Access transistors allow the value stored in the cell to be accessed via the bitlines. Since an SRAM cell can be accessed at or near the speed of a processor, they often are used for register files and small caches [51].

Memory arrays with densities greater than SRAMs are based on the single-transistor, single-capacitor dynamic random-access memory (DRAM) cell, which is shown in Figure 1.2(b). Data is stored as a charge in the capacitor and the cell is accessed via the access transistor using orthogonal wordlines (rows) and bitlines (columns). Asserting the wordline causes the capacitor's charge to be shared with the bitline, which can be sensed as a voltage signal by a differential-mode sense amplifier [51]. Due to their low cost per bit relative to SRAMS, DRAMs are ubiquitous as main memories for most computer systems.

A more exotic class of memory technology involves semiconductor file memory, or simply file memory. This term has been used to describe semiconductor memory used for file storage, such as non-volatile flash memories or solid-state disks [38, 76]. As a more general definition, *file memory* is high-capacity, economical memory that does not guarantee uniform access times to random accesses to storage words. In addition, instead of the usual byte or word addressing, file memory may restrict all accesses to be block-addressable. A hallmark of file memory is



(a) SRAM cell. (b) DRAM cell.

Figure 1.2: A comparison of SRAM and DRAM cells.

to achieve high capacity and low cost at the expense of performance. File memories can be implemented using multiple memory modules [68], wafer-scale integration [21] or fault-tolerant techniques that reduce the overall cost per bit [81, 82]. In this thesis, file memory is understood to be more economical per bit than conventional DRAM.

1.3.1.2 Extended Storage: Revisiting a Good Idea

File memory is suitable for use as extended storage in computer systems.[1] *Extended storage* refers to large amounts of slow and inexpensive memory placed below main memory in the memory hierarchy [45, 53]. The objective of extended storage is to use slow semiconductor memory or other types of storage to reduce the frequency of disk I/O operations. Extended storage does not necessarily imply the use of file memory. As shown in Figure 1.1, legacy mainframe manufacturers had sufficiently large budgets to base main memory on SRAM technology while extended storage used larger quantities of slower and cheaper DRAM. However, extended storage presently is not common since main memories are based on DRAM technology and economical file memory is unavailable. Nevertheless, current systems could benefit from an architecture that uses extended storage with file memory that is less expensive than conventional DRAM. This thesis will evaluate the suitability of file memory as extended storage to quantify its cost-effectiveness relative to DRAM.

---

[1] "Extended storage" represents all terms that describe the same concept with minor architectural variations. Some synonyms include *expanded storage*, *solid-state disk (SSD)*, *memory extensions*, *external memory* and *external storage* [4, 53, 77].

1.3.2 Fault Tolerance

To improve the cost-effectiveness of a memory technology, various techniques of fault tolerance are employed to improve yield. While yield issues are common for DRAM technology, file memory yield issues have received little attention.

1.3.2.1 DRAM Yield

Successful high-density DRAM technology depends on effective management of yield issues. To improve manufacturing yield, DRAM designs often feature redundancy and error-correction codes (ECC) [65]. During device verification, rows and columns that contain defective cells are replaced with spare rows and columns, respectively. Error-correcting codewords typically use a combination of data bits and check bits to detect up to two errors and to correct single-bit errors [65].

1.3.2.2 New Ideas for File Memory

DRAM remains expensive relative to magnetic and optical storage media. This cost disparity could be reduced if it becomes possible to overcome the requirement of high yield rates, since 100% of the nominal capacity of a DRAM must be functional. One approach for reducing the overall cost per bit increases DRAM density by storing multiple bits per cell [78]. Other work proposes that DRAM designs should be augmented with circuitry and non-volatile memory to mark bad blocks to improve yield and reduce the overall cost per bit [81]. In Chapter 3, a system-level approach introduces the idea that bad block marking is more effectively managed by the operating system when file memory is employed as extended storage. This involves adapting existing dynamic memory allocation algorithms for bad block management.

1.3.3 ESDC Design

Extended storage can reduce the frequency of I/O operations if it is added as a new stage in the memory hierarchy of a computer system. When placed between main memory and disk, extended storage can be designed to function as a page-addressable, general purpose disk cache for both file I/O and demand paging (see Chapter 4). This architecture should be designed to facilitate accurate acquisition of experimental results.

1.3.3.1 An Experimental Platform

Evaluating the performance of file memory as extended storage involves the creation of an experimental platform. As discussed in Chapter 5, a methodology based on simulation of a memory hierarchy model was rejected due to the complexities of modern operating systems and the level of the memory hierarchy under investigation. Therefore, an open-source operating system kernel, Linux 2.4.18, was enhanced to include extended storage as a disk cache. While more effort is necessary to decipher the intricacies of a poorly-documented, open-source operating system, the results can predict file memory performance more accurately than a simulation model.

1.3.3.2 New Ideas for File Memory Evaluation

File memory can be evaluated using conventional DRAM. It is relatively straightforward to create a model of file memory by defining a portion of standard DRAM as extended storage and applying appropriate performance penalties to all extended storage accesses. Defective areas of file memory can be modeled via the operating system or by simply reducing the size of extended storage.

The design of ESDC is based on the principle of minimizing implementation effort. Existing data structures and virtual disk caches are adapted to support ESDC instead of introducing an independent design. This conservative design approach is essential for implementation robustness. Adequate verification of the implementation should precede the experimental evaluation.

1.4 Applications of Extended Storage

Using file memory as extended storage in a memory hierarchy has a variety of potential applications. The most obvious use of a modified memory hierarchy is to improve the performance of computer systems, such as workstations and servers. The performance of portable devices could benefit from the introduction of extended storage using file memory even in the absence of magnetic media. Extended storage disk cache design could expand the currently limited market of downgraded memories. Finally, recent research into new storage technologies could create devices that fill the gap in the memory hierarchy between semiconductor devices and disk, in terms of performance and cost.

1.4.1 Personal Computer Systems

Introducing extended storage into the memory hierarchy of current workstations would bridge the performance gap with minimal impact. A Linux operating system is modified by a small kernel patch that introduces subtle changes to the memory hierarchy but preserves operating system stability. However, there are several limitations of the current ESDC implementation. It is only available for the Linux operating system running the 2.4.18 kernel. As well, ESDC is designed for 32-bit architectures, so future versions for 64-bit operating systems will require additional design effort. Finally, thorough testing of ESDC is necessary before recommending its use on servers, systems with multiple processors, or other critical applications.

1.4.2 Portable Devices

Extended storage using file memory is particularly suitable for portable devices that run simplified or embedded operating systems. Devices such as portable digital assistants (PDAs) often suffer from limited storage capacities due to poor hard disk portability, high power consumption, and high cost of non-volatile memories. To improve PDA performance, a memory hierarchy could be introduced, where DRAM is backed by file memory, which is then backed by flash memory. Since some PDAs currently support the Linux operating system [10], extended storage could be utilized to provide a cost-effective method of increasing the capacity of virtual disk caches.

1.4.3 Downgraded Memory

Extended storage depends on the availability of a memory product that is less expensive than conventional DRAMs. Fortunately, various markets currently exist for *downgraded DRAMs* [26]. These chips are also known as *audio RAM (ARAM)*, *reduced spec RAM*, or *toy grade RAM*. These chips have failed the manufacturer's final testing stage but still have functional areas of memory. They are economical since they are available at a fraction of the cost of conventional DRAMs. They also are used for applications that can tolerate various amounts of single-cell errors, such as memory for portable digital audio. Products that do not require high sound quality can tolerate a larger number of clustered errors, such as answering machine memories or toy applications. Once supported by a modified operating system and optional hardware enhancements, various types of downgraded memory could be exploited as extended storage.

1.4.4 MEMS Storage Technology

Experimental technologies, such as micro-electromechanical systems (MEMS) or nanotechnology, could soon result in the commercialization of new classes of storage devices. As predicted in [16], significant gaps in the memory hierarchy of computer systems could be filled by probe-based MEMS-actuated chips. Once such technology matures, hierarchy gaps in latency, capacity and price could be reduced. If successful, devices based on nanotechnology could function as alternative media for file memory. The concept of using this technology as extended storage to improve performance could be feasible depending on the access times relative to semiconductor memory.

1.4.5 Multilevel DRAM

Recent research has investigated the concept of increasing semiconductor memory density by storing multiple bits per memory cell [85]. Since the cell area is not increased, multilevel DRAM technology increases storage capacity more cost-effectively than conventional DRAM technology. However, the access times of multilevel DRAMs are inferior due to complex voltage sensing operations [78]. Therefore, multilevel DRAMs may be suitable for filling the gap in the memory hierarchy between semiconductor devices and disk.

1.5 Conclusion

Extended storage has the potential to improve the performance of inexpensive desktop computers and servers with minimal increases in cost. Extended storage is not present in modern systems due to the absence of a suitable memory product with appropriate cost and performance characteristics relative to DRAM and disk. However, file memory research may create a new type of memory technology that, while slower than conventional DRAM, is substantially more cost-effective. This would permit transferring the concept of extended storage from expensive supercomputers to the modern personal computer.

> Since extended memory is less expensive than main memory, the cost-effectiveness of caching can be improved by choosing a small main memory and a large extended memory buffer.
>
> — Erhard Rahm, 1992
> *Describing extended storage on*
> *an IBM 3090 mainframe* [53]

Chapter 2

Concepts

2.1 Introduction

For years, computer architects have examined the problem of the access time gap between memory and disk. One way to accomplish this is to employ file memory as extended storage in the form of a *disk cache*. Disk caches are typically used to cache or buffer explicit or implicit I/O operations. If a cost-effective disk cache can capture a large number of I/O operations with access times better than disk, then it has the potential to improve system performance [63]. Situated at low levels of a memory hierarchy, disk caches primarily allow modified data blocks to be buffered before writing them to disk. This allows blocks to be arranged so that they can be written to disk as efficiently as possible [56]. Large extended storage capacities permit larger disk caches that can assist with virtual memory operations such as demand paging. Larger disk caches also intercept a greater number of read requests and reduce the number of writes to lower levels of the memory hierarchy. In spite of the performances advantages, reliability issues must be considered when disk caches are based on semiconductor media. However, extended storage does not necessarily require semiconductor memory; fast mechanical media functioning as disk caches have been shown to improve the overall performance of a hierarchy of storage devices. This chapter will review how previous disk caches used types of memory and storage media that were slower and more cost-effective per bit than conventional main memory.

Table 2.1: Memory Hierarchy Characteristics [25]

Hierarchy Level	**Technology**	**Access Time (ns)**[a]	**Typical Size**
Processor Registers	SRAM	0.25–0.5	< 1 KB
Processor Caches	SRAM	0.5–25	< 16 MB
Main Memory	DRAM	80–250	< 16 GB
Hard Disk	magnetic media	5,000,000	> 100 GB

---

[a] Access times are typical values for 2001

2.2 File Memory as Extended Storage

The performance of a computer system frequently is improved by simply increasing main memory size. While this can reduce the number of I/O requests, it ignores the fact that there is a dramatic gap between the performance of DRAM and disk. This performance gap can be addressed by placing slower and less expensive memory between main memory and disk as extended storage. File memory is a good candidate technology, since it is both slower and more economical than DRAM. However, it must be demonstrated that economical file memory is feasible before extended storage can be introduced into modern computer systems.

2.2.1 The Performance Gap

Throughout its history, the access time of disk technology has improved relatively slowly, even though disk capacity has increased exponentially. The access latencies of DRAM memory and disk differ by four to five orders of magnitude while other adjacent levels typically differ by less than one order of magnitude [83]. The disparity between the access time of disk relative to other levels of the memory hierarchy is shown in Table 2.1. This access time gap causes processor under-utilization and limits system response time [36, 55].

Numerous methods have been used to address the performance gap between disk and memory speeds. Operating systems use a host of techniques to improve performance by avoiding unnecessary disk I/O. Since I/O data rates are limited by the mechanical speeds of disks, problems that were once CPU-bound have become I/O bound [40]. As well, programs that use large data structures are sensitive to virtual memory paging due to the widening disparity between processor speed and disk I/O performance [45]. There are a variety of efforts to address the access time gap by reducing I/O latency, such as increasing disk platter rotation speeds, improving disk scheduling algorithms, and increasing I/O bus speeds [36]. A wide range of different types and sizes of memories are used either as replacements for disks or as caches in the I/O data path [53, 63]. According to Fujiwara and Tanaka, yield improvements could facilitate the introduction of large memories to fill the speed-gap between main memories and disk [21].

In [28], it was noted that the history of the development of memory and disk technology are similar and could offer insight into future trends. Memory interleaving was developed for early computer memories to improve data throughput. Later, processor cache memories solved the problem of main memory latencies and memory interleaving became obsolete. RAID (redundant arrays of inexpensive disk) architectures prevent data loss due to failing media but also increase disk I/O bandwidth because multiple disks are accessed in parallel. Even though redundancy is absent, interleaved memories are analogous to RAID systems in terms of the improvements in throughput. Disk caches, analogous to processor caches, attempt to address disk I/O latencies. However, Hu and Yang observe that existing disk caches are orders of magnitude smaller than disks due to the large disparity in cost per bit between DRAM and disk media [28]. Therefore, traditional disk caches have not been as successful as caches for main memories. If it became possible to fabricate a new type of memory technology with a lower overall cost per bit, it could be used as a large, slow, page-addressed cache between DRAM and disk in the memory hierarchy.

2.2.2 Economical File Memory

File memory must be fabricated more economically than commercially available DRAM before it can become feasible as extended storage in the memory hierarchy.

Cost models have shown that cost-effective file memory is feasible if two DRAM design requirements are relaxed [81, 82]. First, file memory is addressed as large blocks, such as 4-KB pages, instead of fast byte-level or word-level random access. Second, the requirement that 100% of the nominal memory capacity must be addressable is waived for extended storage. In addition to conventional error correction and redundancy techniques, file memory has a non-contiguous address space since faulty memory blocks are avoided. Marking bad blocks has the potential to further reduce the average cost per working bit and is similar to the remapping of flawed magnetic disk sectors.

Stapper *et al.* discovered that combining memory redundancy techniques with error-correcting codes (ECC) produces a DRAM chip with significantly better fault tolerance than the fault protection provided by either scheme independently [65]. A similar synergistic effect is realized when ECC and redundancy are combined with bad block marking [82]. Marking blocks as bad implies marketing a chip with near-nominal capacity, which is common practice for disks [58]. Such a chip can not be used as main memory since a contiguous range of physical memory addresses are required in some situations, such as DMA operations. Furthermore, a constant access time to any cell may not be possible due to some suggested mechanisms for bypassing bad blocks. A map of defects can be stored in a small, non-volatile memory and memory accesses are redirected to avoid bad blocks. Bad block marking can be implemented as peripheral circuitry within the DRAM chip and may require the cooperation of system software [82]. This approach has a number of disadvantages, which limit file memory's potential performance and increase costs. As will be discussed in Chapter 3, these problems can be addressed by operating system modifications that prevent bad blocks from being allocated for use.

Using yield and cost models, it was shown that bad block marking offers improved yield that results in a lower overall cost per working bit [81]. Assuming that ECC, redundancy and bad block marking are used during early production of a high-density DRAM chip, then the device can be sold as extended storage. As the memory process matures, the redundancy can be used to achieve perfect media and the device can be sold as conventional DRAM [81]. By employing an accurate model of file memory as extended storage, this thesis will quantify file memory's cost-effectiveness relative to conventional DRAM.

2.3 Extended Storage Architectures

Semiconductor extended storage may be found in one or more levels of a memory hierarchy. Extended storage primarily is used as cache to improve I/O performance and as cache of demand paging backing store. It is important to note that extended storage is not just a method of adding memory to a system; additional general-purpose system memory will improve I/O performance provided that the memory is used for functions related to I/O [63]. Due to the wide variety of uses of extended storage memory, this section will begin with an overview of several different types of uses of extended storage in the memory hierarchy. There are a number of disadvantages of having a hierarchical extended storage architecture. In response to such deficiencies, one proposal uses extended storage as a direct memory extension rather than a stage in the hierarchy [45].

2.3.1 Extended Storage Hierarchy

Memory-based extended storage, with a wide range of capacities, has been used extensively as caches in I/O datapaths. It is necessary to determine where a disk cache should be placed along the datapath between the processor and the lowest level of the memory hierarchy (see Figure 2.1). Some extended storage disk caches are subsets of existing levels of the memory hierarchy while others function as distinct levels. Each type of extended storage disk cache requires a brief explanation.

A *device cache* is a relatively small memory embedded into a device such as a disk [63]. Devices frequently use embedded caches to assist with read-aheads and reduce I/O latencies. For example, a 30-GB Western Digital Caviar drive uses a 2-MB I/O buffer [80]. Volatile caches can only improve read performance. However, the use of non-volatile memory (NVRAM) enables staged writes that minimize disk head movements. Intelligent interfaces use device caches to accept data regardless

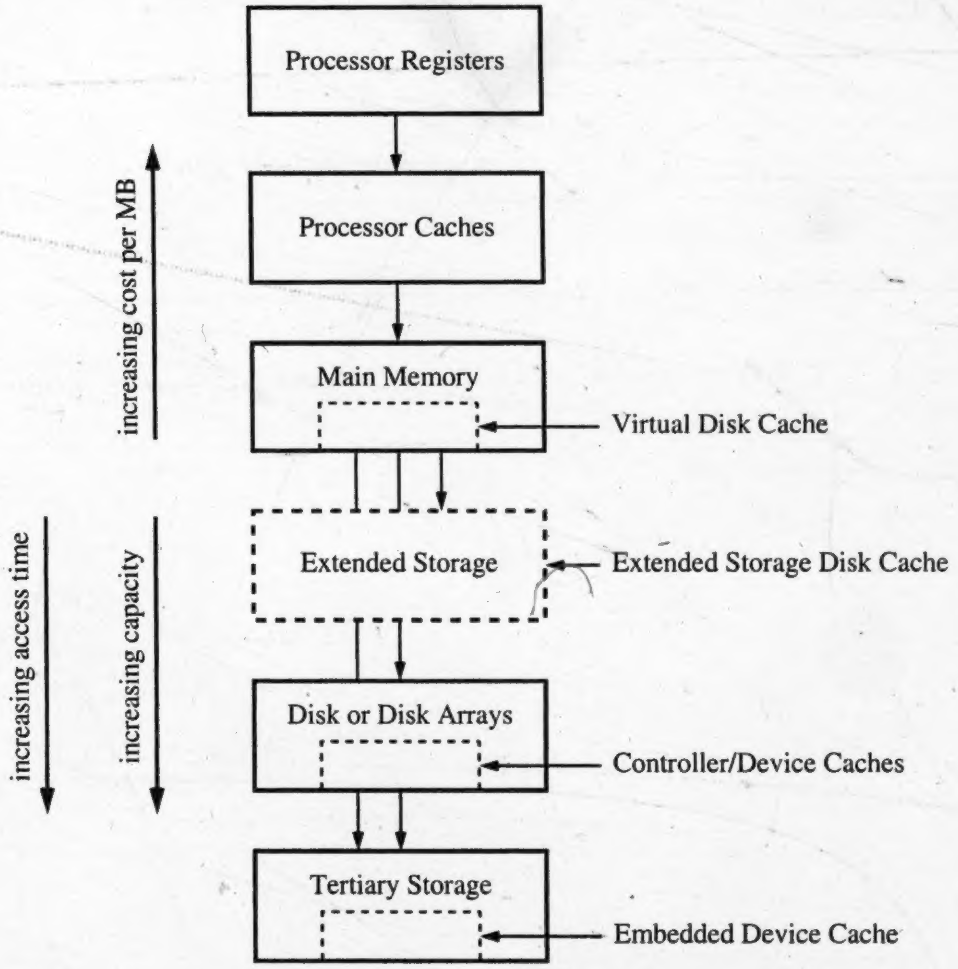

Processor Registers
Processor Caches
Main Memory
Virtual Disk Cache
Extended Storage
Extended Storage Disk Cache
Disk or Disk Arrays
Controller/Device Caches
Tertiary Storage
Embedded Device Cache
increasing cost per MB
increasing access time
increasing capacity

Figure 2.1: A memory hierarchy with possible locations of extended storage [36, 53].

of the position of the disk write heads. That is, the host is no longer required to synchronize its data transfer to the drive according to the position of the heads [2].

A *controller cache* is a disk cache in a multi-disk controller that helps the controller manage functions such as command queuing, seek ordering and DMA transfers while allowing the processor to proceed with other tasks [36]. Access times are usually limited by the speed of the I/O bus and the controller [53]. Controller caches are a general purpose solution that do not require any operating system changes and are independent of other components of the system [63].

Extended storage is slow, page-addressable memory that can be located below main memory in a memory hierarchy. Its position above disk media is conceptual. That is, in some architectures, pages cannot directly migrate from extended storage to disk without passing through main memory due to the absence of a separate extended storage controller [53]. As well, the memory used for extended storage should be more cost-effective than standard DRAM, and typically has slower access times. For example, Rahm indicated that main memory is about twice as expensive as extended storage for the IBM 3090 mainframe [53]. Since it is economical, using extended storage as a cache to improve I/O performance is more cost-effective than simply increasing the size of main memory. A virtual memory manager can use extended storage to cache pages for faster demand paging. A page fault can be serviced from extended storage memory rather than from backing store on disk [45]. Extended storage could serve only as a fast paging device, but this thesis will focus on the uses of extended storage memory as a general-purpose stage in the memory hierarchy. Extended storage that is used to cache I/O operations as well as virtual memory pages will be referred to as an *extended storage disk cache (ESDC)*.

A *virtual disk cache* is a page-addressable operating system cache that uses a variable number of free memory pages to cache a variety of I/O operations. Since this cache is a subset of main memory, it is not a physically distinct hierarchy level. The most common type of virtual disk cache is a page cache that caches pages associated with files on disk [36, 53]. Other virtual disk caches include buffer I/O caches [63] and swap caches [8]. Page caches form the foundation for ESDC architecture and are examined in detail in Chapter 4.

Solid-state disks (SSDs) use NVRAM, such as flash memory, as a storage device instead of slow magnetic media.[1] Solid-state disks provide benefits of portability, reliability and durability. Note that an SSD does not have an ideal conceptual location in a memory hierarchy, as illustrated in Figure 2.2. It could reside below main memory and extended storage since it has lower performance than DRAM [53]. For example, implementations that use battery-backed SRAM or DRAM to

---

[1] Cray used the same SSD acronym to describe a different type of memory (see Section 2.3.4).

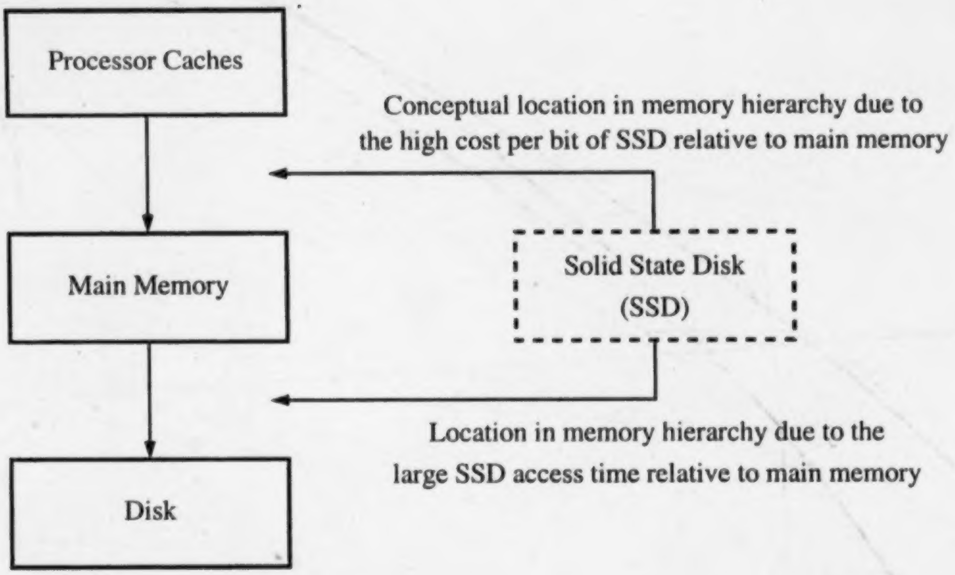

Processor Caches
Main Memory
Disk
Solid State Disk
(SSD)
Conceptual location in memory hierarchy due to the high cost per bit of SSD relative to main memory
Location in memory hierarchy due to the large SSD access time relative to main memory

Figure 2.2: Conflicting conceptual locations of non-volatile solid-state disk in a memory hierarchy. This scenario assumes that the size of SSD and main memory are equivalent.

improve performance tend to place non-volatile memory on a slow I/O bus [47]. However, an SSD device does not fit below DRAM-based main memory due to its limited capacity and relatively high cost per bit. Large non-volatile memory sizes currently are not feasible due to the prohibitive cost per megabyte of NVRAM relative to both disk storage and conventional DRAM [5].

A comprehensive evaluation of disk caches in all of these hierarchy levels (except extended storage) was completed by A. J. Smith [63]. Smith concludes that a virtual disk cache in main memory should eliminate most overhead and provide the best performance improvement. Smith points out that disk caches in memory, rather than in the controller or device, offer the advantage of being shared among multiple I/O devices. Smith also evaluated how the miss ratio is affected by various disk cache capacities, ranging disk cache size from a few kilobytes to several hundred megabytes. Disk caches between 2 MB to 8 MB were suitable for an IBM 370/168, but it was indicated that larger capacities would be required for faster processors since they access more data per unit time [63]. These results suggest that a distinct memory hierarchy stage, consisting of inexpensive file memory, should be placed below main memory and above disk. As well, Smith's work supports the argument for large extended storage disk caches because modern processors are three orders of magnitude faster than an IBM 370.

2.3.2 Disadvantages of Extended Storage Hierarchies

Using extended storage as a disk cache has a number of disadvantages. First, caching can experience diminishing marginal returns as the size of the cache grows larger, as discussed in [55]. However, this work relies on traces that do not involve kernel file system activity, such as paging from executable images. Second, a variety of common file system operations cause cache pollution—reading large files, system backups, file system searches, and disk maintenance utilities. Cache pollution can be solved by designing more intelligent applications or incorporating process analysis algorithms within the kernel [79]. Third, transferring data between disk and main memory can be very complex because a host of various techniques are used to schedule I/O operations, manage metadata and reduce I/O latency [79].

An evaluation of extended storage architectures by Li and Petersen raised questions about placing extended storage in a memory hierarchy in their *GigaSUN* architecture [45]. In this work, two different extended storage architectures were compared. The first architecture is similar to ESDC, where inexpensive and slow memory modules function as a large cache between main memory and disk. The alternative architecture places slower extended storage memory at the same level as main memory in the memory hierarchy. This method requires a secondary memory bus for the slower memory. The architecture was implemented in hardware with 32 MB of main memory and 512 MB of extended storage. The access time of extended storage (300 ns) in the GigaSUN project is twice the access time of main memory. The experiments summarized in [45] show that the hierarchical architecture performed better than the baseline system, but the alternative architecture outperforms the extended storage hierarchy for memory-bound applications.

The conclusion made in [45] that their alternative architecture is superior re-

Table 2.2: Extended Storage Specifications for Four Architectures [45, 50, 77]

System	Main Memory		Extended Storage	
	Max Capacity	Access Time	Max Capacity	Access Time
GigaSUN	32 MB	300 ns	512 MB	600 ns
IBM 3090	512 MB	300-350 ns[a]	4096 MB	> 350 ns[b]
ES/9000	2048 MB	300-350 ns[a]	8192 MB	> 350 ns[b]
Cray Y-MP	1024 MB	15 ns	4096 MB	50 ns[c]

---

[a] On average, one 16-byte doubleword is transferred per processor cycle, according to Prasad and Savit [50, p. 10, 244].

[b] 75 μs per 4-KB page transfer to main memory over separate datapaths. One 16-byte doubleword is transferred every 4 clock cycles [50, 53].

[c] A 4-KB block in SSD can be directly accessed in 25 μs, so the access time for a 64-bit word is 50 ns [49].

quires several clarifications. First, the authors indicate that the paging mechanism used by the hierarchical architecture was implemented as a user process that has more significant fault processing overhead than the kernel pager used for the alternative architecture. Second, the alternative architecture relies on a dedicated secondary memory bus; the resulting performance gains have to be weighed against the additional expense of adding a duplicate memory bus. Other studies argue for a hierarchical approach. For example, Rahm shows that cost-effective extended storage is most effectively used as a write buffer or as an additional level in a memory hierarchy [53]. As well, extended storage in the memory hierarchy was championed by the architects of legacy mainframes.

2.3.3 Historical Study: The IBM 3090 Mainframe

The IBM 3090 was one of the first architectures to introduce a form of page-addressable extended storage known as expanded storage [77]. Expanded storage is semiconductor memory that allows synchronous block transfers of 4-KB pages between itself and central storage (i.e. main memory) [17]. It is important to note that expanded storage in the 3090 is not simply additional main memory. Expanded storage is slower and less expensive than main memory [50, 86]. In the 3090, ex-

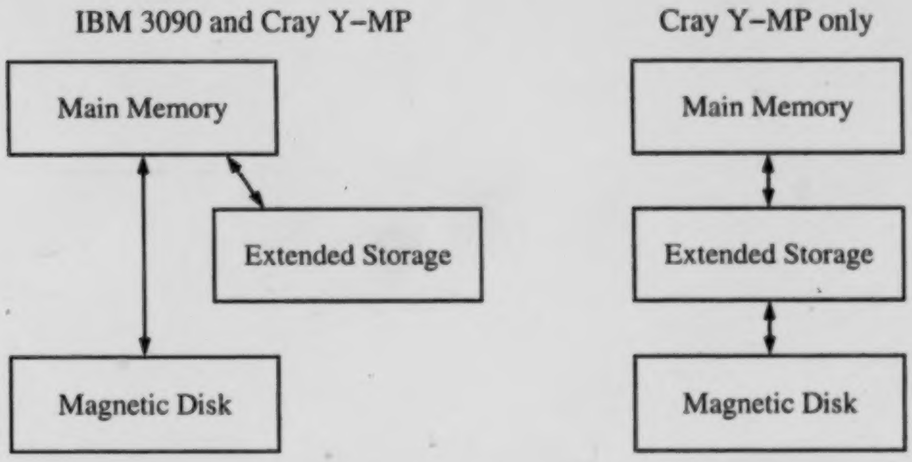


Figure 2.3: The IBM 3090 and Cray Y-MP memory hierarchies [49, 77].

panded storage is used as paging space, virtual I/O and other types of data spaces [86]. Expanded storage is page-addressed, which reduces the required width of the address bus and permits substantial expansion potential. However, pages in expanded storage are accessed only via main memory (see Figure 2.3). That is, pages must be transferred from expanded storage to main memory to be accessed by the processor or *migrated* to disk [15, 17, 45, 53]. According to Prasad and Savit, this restriction reduces cache interference since data copied between main memory and expanded storage do not displace cache contents [50, p. 203]. This data transfer proceeds over separate data paths so that a 3090 can transfer a 4-KB page in 75 μs. Expanded storage in Enterprise System Architectures (ESA), such as the IBM 3090 and the ES/9000, also functions as a cache for demand paging. Least-recently used (LRU) replacement is used between main memory and expanded storage and expanded storage and disk [50, p. 256]. Specifications of the memory subsystems of these architectures are summarized in Table 2.2.

2.3.4 Historical Study: The Cray Y-MP Mainframe

Another example of extended storage is the solid-state disk found on Cray mainframes such as the Cray Y-MP. Providing up to 512 million words (4 GB) of "external memory", Cray SSDs are analogous to expanded storage on the IBM 3090

[4]. Once again, this architecture demonstrates how main memory is more expensive and faster than extended storage. The Cray Y-MP uses bipolar SRAM as main memory with a 15-ns access time while the SSD uses DRAM technology [31]. The SSD can be accessed in three ways. First, it can be used as a logical disk for frequently accessed files. However, this is not the best use of an SSD, since it makes use of device drivers optimized for disks rather than for semiconductor storage. Second, a Cray SSD can be used as extended storage in a manner similar to IBM's expanded storage. Third, the SSD can be configured as a distinct level of the memory hierarchy that resides between a virtual file cache in main memory and physical disks (see Figure 2.3). The last alternative has been shown to improve workload performance by a factor of four on the Y-MP [49].

2.3.5 Summary

In general, designers of legacy systems had sufficient budgets to design systems with expensive SRAM as main memory and used more cost-effective DRAM as extended storage. Several generalizations can be made from the specifications listed in Table 2.2. First, the capacity of extended storage was larger than the main memory size by at least a factor of four. Second, the access times of the memory used for extended storage were usually a factor of two or three greater than those for main memory. In a performance evaluation comparing the various types of extended storage, Rahm found that extended storage was more effective than device caches [53]. These historical relationships can help guide the design of modern file memory so that it can serve as effective extended storage.

2.4 Extended Storage Disk Architectures

Extended storage memory has been utilized as a variety of types of disk caches since it is slower and more economical than conventional DRAM. However, extended storage need not be restricted to semiconductor media. It has been shown that small, fast disks functioning as caches can improve the performance of systems using large and slow disks. In addition, disks can be used to cache I/O to slower storage media

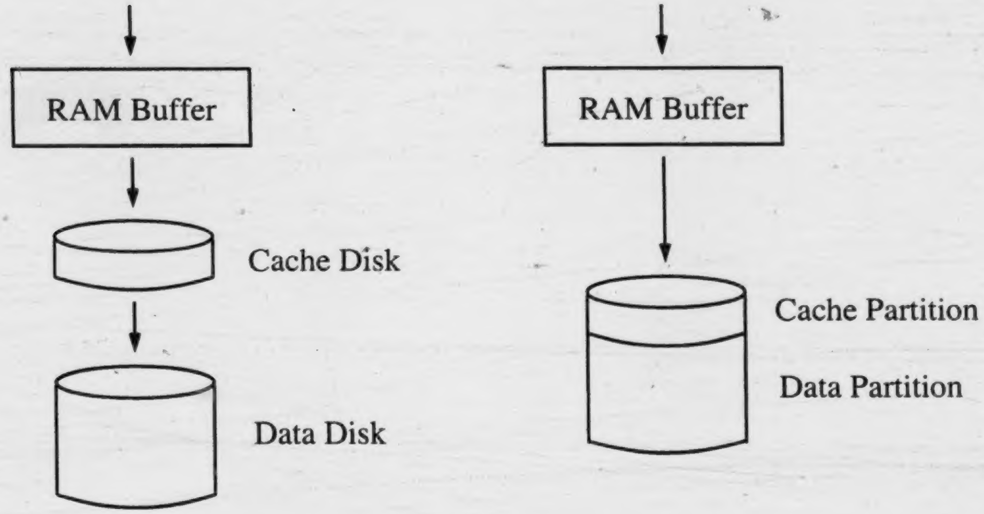


Figure 2.4: The disk caching disk (DCD) hierarchy [48].

such as optical disks, tapes, or distributed storage. An examination of extended storage disk architectures can help to reveal potential methods of improving storage I/O performance through caching.

2.4.1 Disk Caching Disk

To reduce disk access latencies, a proposed disk hierarchy uses a small, fast log disk as a cache for a data disk [28, 29]. As shown in Figure 2.4, the disk caching disk (DCD) architecture resembles a cache hierarchy. The cache disk, a physical disk or a small partition of a larger disk, is an extension of a memory-based disk cache. A log-structured file system is used to improve performance (see Section 2.5.2). Writes are queued in a small RAM buffer before being written to the cache disk. Data is transferred from the cache disk to the data disk when the latter is idle.

The DCD architecture attempts to address the performance gap in the memory hierarchy (see Section 2.2.1). Since the capacity of the cache disk—tens of megabytes—is larger than available memory, it can capture the temporal locality of disk I/O transfers. It has been shown by simulation that the average response time for write operations was reduced by up to two orders of magnitude [28]. Another advantage of this method is reliability since the cache disk is inherently nonvolatile. An implementation of DCD involved creating a DCD device driver for Sun's Solaris operating system [48]. This unoptimized logical DCD reduces program execution times by factors ranging from 2 to 6. Finally, since the DCD implementation is a device driver, no operating system modifications were necessary. This is noteworthy since it is advantageous to minimize the impact that new memory and storage hierarchy implementations have on an operating system.

The performance results of DCD are encouraging, but this work raises several questions. First, the size of the kernel buffer cache is not reported in [48]. Second, the authors acknowledge that it is not feasible to make the cache disk much faster than the order-of-magnitude larger data disk since both are devices with mechanical limitations [29]. As well, reliability is dependent on suitable cache disk fault-tolerance. Instead, this thesis proposes using a large quantity of inexpensive extended storage memory instead of a cache disk. Even if such memory is very slow relative to main memory, it is orders of magnitude faster than a cache disk.

2.4.2 Tertiary Storage

Employing disks as caches is a well-known technique for managing some mass storage systems. Such systems feature tertiary storage, which is media that is slower than disk such as tape drives or some form of optical storage. Some file systems have been designed so that files on slow tertiary storage are cached by disk media [43, 52]. An example of one of these file system architectures is HighLight. This design is based on a log-structured file system (LFS) [56]. HighLight's performance depends on a tertiary storage cache located on the disk array. Possible cache replacement policies include LRU, random and working set. Files that are less likely to be used are migrated to tertiary storage [43].

2.5 Architectural Support for Extended Storage

It is informative to discuss how ESDC is affected by several different architectural innovations. Using extended storage in a memory hierarchy is promising since large virtual disk caches are common in operating systems to improve overall system performance. It has been shown that a compressed virtual disk cache offers performance advantages since it permits more data to be cached in main memory. Log-structured file system design relies on large disk caches that reduce the number of read requests that access the disk. Some file systems have even been designed to reduce disk media usage.

2.5.1 Compressed Caching

One method of increasing usable memory capacity and system performance is to compress virtual memory pages within a new level of the memory hierarchy. Compressed caching improves performance without adding more physical memory to a system. Although previous work produced mixed results, compressed virtual memory is becoming increasingly attractive as processor speeds increase faster than disk speeds [83]. As detailed in Kaplan's doctoral dissertation, compressed caching divides the virtual disk cache of an operating system into an uncompressed cache and a compressed cache [35]. When the uncompressed cache reaches capacity, pages are evicted to the compressed cache. Pages are migrated to backing store when the compressed cache becomes full. The compressed cache size is dynamically adjusted using an adaptive technique that monitors recent program activity to determine which pages should be compressed. The processing overhead incurred when compressing pages does not affect overall performance since compressed caching reduces paging costs from 20% to 80%, which results in net performance improvements [83].

Compressed caching has been implemented as a patch to the 2.4.18 Linux kernel. This is an implementation of adaptive compressed caching that tests workloads with varying degrees of compressibility. The reported compression ratios ranged from 35.5% to 86.5%. All tested workloads that were run under memory pressure show that compressed caching offers performance improvements of up to 171.4%. When the memory resources of the system are not stressed, compressed caching has negligible overhead (up to 0.39%) [10].

The design of compressed caching has several implications for the design of ESDC. First, it was found that the size of non-compressed memory is directly related to overall system performance [10, 11]. This indicates that large virtual disk caches are necessary to accommodate the performance gap in the memory hierarchy. Instead of only serving as a virtual swap space for pages backed by swap, the compressed cache implementation also compresses page cache pages (pages backed by files on disk). If only virtual memory pages are cached, a performance slowdown for a number of benchmarks is observed [10]. The decision to consider both virtual memory pages as well as page cache pages was made independently during the design of ESDC. However, ESDC only caches virtual memory pages rather than functioning as a paging device.

2.5.2 Log-Structured File Systems

New file systems have been designed to exploit the properties of memory-based disk caches. The log-structured file system (LFS) is one example of a file system design that relies on the fact that files are cached in memory [56, 59]. The assumption is that increased memory sizes will permit a larger number of memory-cached files, so fewer read requests will make it to disk. However, Roselli and others argue that this assumption also depends on the workload and write delay [55]. Nevertheless, a log-structured file system is optimized for disk writes. All writes to disk are indexed in a sequential log structure on disk. Crash recovery is simpler for an LFS since only the log must be scanned after a crash. LFS is not the most successful file system due to the introduction of additional complexity to an operating system and poor disk utilization [28].

2.5.3 Conquest File System

Instead of relying on disk caches to improve performance, some file systems are designed to exploit large memory capacities. The Conquest file system uses battery-backed RAM and disk as a hybrid file system [79]. In this work, most file system functions are performed in memory, while disk is reserved for large file storage.

Table 2.3: Impact of Disk Cache Write Policy on Performance and Reliability [47]

Policy	Reliability	Performance	Description
write-through	excellent	poor	writes to disk are immediate and synchronous
asynchronous write-through	good	fair	writes initiated immediately but do not block process
delayed write	fair	good	writes initiated after a delay
write-back	poor	excellent	capacity-induced writes to disk

Overall performance improvement compared to disk based file systems ranges from 43% to 96%. This work avoids the unnecessary disk management overhead used to support RAM drives and RAM file systems. That is, the Conquest file system achieves 15% faster performance than the *ramfs* file system since the latter is optimized for a device with mechanical limitations. This fact influences ESDC design in that ESDC should be managed as a memory resource rather than as a physical storage device. Section 4.5 has more information about the impact of this decision on ESDC implementation.

The Conquest file system requires extensive modifications to the kernel to avoid management overhead. Conquest is not suitable for extended storage since it is not designed for a combination of fast and slow memory technologies. Following the example set by the Conquest file system, ESDC could be designed with a bias for small files with predictive algorithms to reduce the frequency of situations that cause disk cache pollution. However, such enhancements are beyond the scope of this thesis.

2.6 Reliability versus Performance

Volatile extended storage memories raise questions about reliability since files are temporarily held in memory. As summarized in Table 2.3, the write policy of a disk cache is a trade-off between performance and reliability [47]. Unlike a conventional

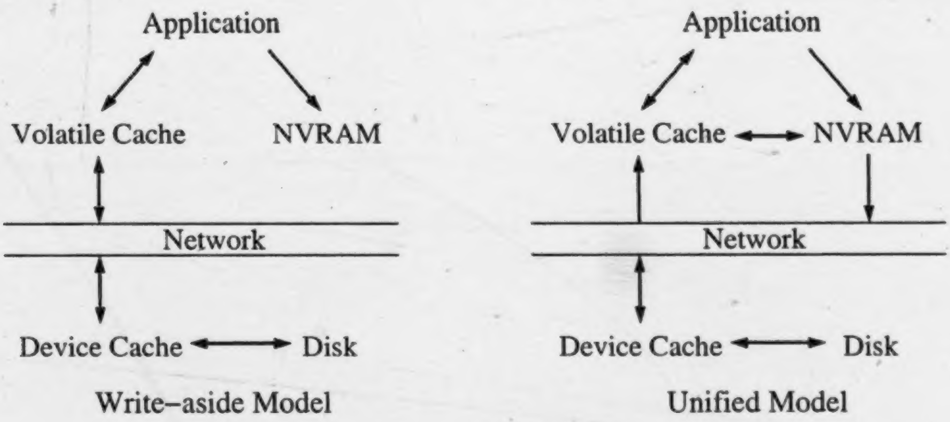


Figure 2.5: Non-volatile RAM cache models [5].

write-back cache, writing data to disk when the disk cache reaches capacity would compromise reliability. Writing pages to the disk immediately would offer high reliability at the expense of operating system performance. As a hybrid solution, periodically writing cache files to disk is used by a number of operating systems [1, 8, 13]. Nevertheless, it is informative to review several efforts that attempt to address the inherent unreliability of memory. One effort introduces a small portion of non-volatile RAM into the memory hierarchy while another project uses different methods to store files reliably in memory.

2.6.1 Partially-Safe Disk Caches

The use of non-volatile memory as extended storage would offer reliability in the presence of power interruption, but it also would dramatically decrease the cost-effectiveness. The cost of NVRAM makes it unsuitable for a memory hierarchy stage located below DRAM. Fortunately, only a small portion of the disk buffer memory would need to be non-volatile. One architecture, shown in Figure 2.5, uses both volatile and non-volatile memory as a distributed file system cache [5]. In the write-aside model, NVRAM stores duplicate copies of dirty data blocks in a volatile disk cache. The non-volatile memory is only read during crash recovery. This model does not perform as well as an integrated model, where dirty blocks reside in the NVRAM and clean blocks can be placed in either type of memory. The latter model is similar to the architecture used in [1]. Since non-volatile memory is approximately five times as expensive as volatile memory, an LRU replacement policy is modified to perform replacements according to the volatility of the buffer.

These models demonstrate that a small amount of NVRAM can improve the reliability of a larger volatile disk cache. However, if file memory is to be used as extended storage, it should be more economical than conventional DRAM. Therefore, a portion of non-volatile memory would have an negative impact on cost-effectiveness due to its high relative cost.

2.6.2 The Rio File Cache

The objective of the Rio (RAM I/O) file cache is to enable memory to store files with the reliability of disk-based storage [13, 14, 47]. In this work, files are cached in a region of memory that forces all accesses to go through an interface to improve the reliability of memory-based file storage.[2] This approach attempts to make a compromise between two ideals: the reliability of a write-through disk cache and the performance of a write-back disk cache.

Even if memory does not suffer from power failures, the authors still regard it as unreliable for file storage. The authors of [14] reason that memory addresses can be written with no explicit protocols, while a disk interface is explicit. Therefore, erroneous software can cause data corruption in memory, but it is unlikely for such errors to corrupt files on disk.

One of the methods requires that a memory device driver manage all memory accesses to the file cache (disk cache). To control accesses to the disk cache, an adaptation of virtual memory protection and code patching of kernel object code ensures that all writes to disk cache addresses are valid. A customized file synchronization mechanism writes dirty data back to disk during a crash so that the reliability of the disk cache is equivalent to the reliability of disk [47]. Such methods would be suitable for enhancing the reliability of an extended storage disk cache. However, simply replacing a virtual disk cache with a dedicated area of memory as extended storage offers superior failure isolation against software errors [53, p. 309]. Therefore, modifying an operating system to incorporate reliability techniques such as those used in the Rio file cache is beyond the scope of this thesis.

---

[2] Maintaining file reliability during power outages is handled by using batteries or uninterruptible power supplies so that the cache can be flushed to disk.

2.7 Conclusion

A large number of solutions have been proposed or implemented to address the increasing access time gap between main memory and disk. Most of the methods involve modifications to the memory hierarchy. Such hierarchy enhancements often are based on the introduction of a new hierarchical level with suitable size, cost and performance characteristics. It has been shown that cost-effective extended storage memory is possible by relaxing both access time guarantees and a contiguous address space. This thesis will show how slow file memory can improve system performance when it functions as an extended storage disk cache for file I/O and virtual memory paging.

To be effective, an extended storage disk cache should be large enough to capture the temporal locality of disk I/O transfers. Previous performance evaluations of extended storage only examined the use of extended storage for transaction processing in mainframes [53]. Instead, the performance of a general purpose extended storage disk cache on modern workstations requires further investigation. Therefore, this thesis analyzes the ESDC size and cost necessary for improving the performance of personal computer systems and inexpensive servers.

> Storage hierarchies have been successfully utilized in addressing the problems of a speed mismatch between the CPU and DASD [disk] and between the CPU and real storage [memory].
>
> — E. I. Cohen, 1989 [15]

Chapter 3

Fault Tolerance

3.1 Introduction

For decades, memory designs have been proposed or implemented with a variety of methods for improving fault tolerance. Such techniques often are useful for yield enhancement as well as tolerance of some types of transient faults [60]. Appropriate applications of fault tolerance techniques to file memories will significantly reduce the overall cost per bit by substantially increasing file memory yield relative to conventional DRAMs.

In this chapter, various techniques of fault tolerance will be analyzed to determine a method that is appropriate for file memory. An overview of fault tolerance techniques for different storage technologies will include disks, caches and flash memories. This survey will be followed by an analysis of previous research involving yield models and marking unusable sections of file memory at the device level. Marking bad blocks with data structures that need to be accessed during every memory access negatively affects file memory performance. To address this problem, a new system-level bad block marking method is proposed, which adapts a memory allocator to mark blocks as faulty. Several proposed implementations illustrate the benefit of allocating faulty blocks instead of incorporating bad block checking into the memory interface. This idea has a major advantage since inexpensive low-grade memory chips that are already on the market potentially could be exploited as file memory for extended storage.

3.2 Methods of Fault Tolerance

The subject of fault tolerance can cover a variety of disciplines. To illustrate the scope of the problem, this section will analyze various techniques for fault tolerance in data storage media other than file memory. First, the methods used to improve the reliability of disk media should be examined since disks are commonly sold with less than nominal capacity. This marketing practice is uncommon in commercial memories, but is the basis for the creation of feasible file memory. Second, research on improving the fault tolerance of caches could provide insight into methods of fault tolerance for an extended storage disk cache. Third, flash memory has recently achieved high capacities using bad block management. Finally, the effectiveness of various techniques, such as error correction and redundancy, can be quantified using yield models.

3.2.1 Fault Tolerance for Disks

File memory can be made cost-effective when blocks are marked as bad, thereby improving yield. This is not normal practice for DRAM since a contiguous address space is required (see Section 2.2.2). However, it is common for disk media; disk drives are sold with less than nominal capacity. Investigating the various methods of handling defective sectors on disks could provide insight into appropriate algorithms for marking bad blocks.

Sector faults can be temporary or permanent. Temporary faults can be repaired using ECC or by repeating the sector access attempt [37]. For example, a drive can make a complex series of adjustments as it repeatedly attempts to access a questionable sector. This involves positional adjustments of the disk head or timing adjustments. Once the information is recovered, the sector may be judged as a permanent fault, marked as bad and remapped to a spare sector [2]. Replacing faulty sectors with spares is knows as *sector skipping*. A remapping table is stored both in the device controller's memory and on disk [37]. Sector skipping is analogous to row and column redundancy in DRAMs, but occurs during operation instead of during fabrication. Another technique to avoid a defective sector is *sector slipping*. The logical block associated with the faulty sector and the blocks following the logical block are "slipped" by one or more sectors. Sector slipping occurs during disk format operations and reduces usable disk capacity [58].

Commercial DRAMs do not offer slipping capabilities since their use as main memories requires a contiguous address space. Therefore, block slipping could be applied to file memory as described in a discussion of capacity degradation of large hierarchical memories in [46]. Note that an ESDC "formatting" process that marks faulty blocks would involve applying a mapping of faulty blocks created during manufacturing.

3.2.2 Fault Tolerance for Caches

Adding stages to the memory hierarchy of a system should not be done without considering the impact of the additional stage on fault tolerance. When transient faults afflict caches, there can be an adverse effect on overall system reliability. For example, a transient fault in a cache may propagate to other lines of the cache during execution of a program [64]. As well, enabling cache memory in a system increases the probability of fault occurrence due to soft errors by two orders of magnitude [30]. Hamming-code based ECC schemes significantly enhance yields, but superior results are possible when ECC is combined with other approaches. For example, a cache miss can be forced when an error is detected in a cache line [30]. A different form of error detection and recovery in caches is that used by redundant systems where hardware voting masks hardware failures [12].

In a *PAD*ded cache, a programmable address decoder can disable faulty blocks of a cache and re-map their references to functional blocks [60]. In conventional caches, a fault-tolerance bit can be associated with a cache block to mark it as faulty. A cache miss will occur for a direct-mapped cache when a faulty block is accessed while the associativity of a set-associative cache is reduced by one. Spare blocks can replace faulty blocks, but a few spare blocks is not effective as the number of faults increases. A programmable address decoder of a *PAD*ded cache

Table 3.1: Minimum Number of Functional Blocks at Shipping and Bad Block Percentages for Toshiba's NAND Flash Memories [71, 72, 73, 74, 75, 76]

Device	Capacity	Block Size	Min Blocks	Max Blocks	Bad Blocks
TC58V64	64-Mbit	8 KB	1014	1024	0.98%
TC58128	128-Mbit	16 KB	1004	1024	1.95%
TC58256	256-Mbit	16 KB	2008	2048	1.95%
TC58512	512-Mbit	16 KB	4016	4096	1.95%
TC58100	1-Gbit	16 KB	8032	8192	1.95%

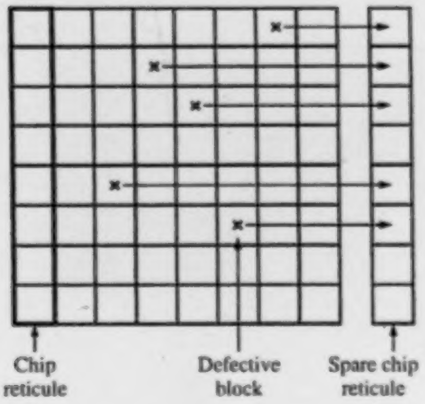
instead remaps a faulty block to a functional block. The addresses that would have mapped to the faulty block will be subject to conflict misses with the functional block. Conflict misses can be reduced by relying on spatial locality and selecting a faulty block from an entirely different area of the address space. The results of this technique show that the method offers the most improvement in fault-tolerance for direct-mapped caches and caches with low associativities. This is not an appropriate solution for a fully-associative disk cache like ESDC. As will be discussed later, a fully-associative cache can be made fault-tolerant by simply marking or avoiding faulty blocks.

3.2.3 Fault Tolerance for Flash

Flash memory is commonly used in non-volatile semiconductor file memory devices.[1] Two types of flash memory technologies include NAND and NOR. NOR flash memories have low capacities and slow write and erase operations. NAND flash technology features higher capacities and performance, but at the expense of lower reliability than NOR flash devices [69]. Therefore, managing bad memory blocks is necessary for products—such as Memory Stick modules and CompactFlash cards—that are based on NAND flash technology. For example, a patent was issued for a bad block management system for flash memory in 1998 [23]. This mechanism detects blocks with bad areas and remaps the data to spare blocks with-

---

[1] While some flash devices are marketed as file memory [76], this type of file memory usually does not have a lower cost per bit than conventional DRAM.



(a) Set-associative mapping.

		Weight Vector			
		00	01	10	11
Address	00	00	01	10	11
	01	01	00	11	10
	10	10	11	00	01
	11	11	10	01	00

(b) Address permutation.

Figure 3.1: Wafer-scale RAM with fault tolerance [21].

out losing data. It also provides information to a flash file system regarding the location of the contents of a desired data block.

Using a technique similar to disk fault tolerance, some NAND flash products ship with bad blocks mapped out [54, 69]. For example, Toshiba markets several NAND flash chips that ship with some blocks marked as bad [76]. As shown in Table 3.1, for nominal capacities ranging from 64 Mbit to 1 Gbit, the worst-case number of usable cells is between 98% and 99% of the nominal capacity. Since entire blocks are marked as bad, the functional cells within those blocks are not accessible. If each of the 160 blocks in the 1-Gbit device have a single faulty cell, then 2621280 functional cells become unaccessible since they are members of bad blocks. That is, Toshiba sacrifices 1.95% of the total potential capacity for a cost-effective commercial product. Disabling relatively large blocks is the basis of page-level bad block marking discussed later in this chapter.

3.2.4 Wafer-Scale File Memory

A wafer-scale file memory described by Fujiwara and Tanaka uses a combination of techniques for fault tolerance [21]. One of the techniques is set-associative mapping, which is used to replace defective memory blocks with spare blocks. As shown in Figure 3.1(a), only one faulty block in each row of the chip array can be replaced by a spare one in the same row. When the number of faulty blocks in a row is larger than the number of spare blocks in that row, the address of the faulty block may be transformed logically to recruit unused spare blocks on other rows. This transformation, called address permutation, involves adding a constant k-bit weight vector to the k-bit block-address of the chip with bit-by-bit exclusive OR operation. Figure 3.1(b) shows an example of address permutation for a 2-bit block address. Additional weight vectors are added to the original address until there is at most one faulty block in a row. In addition to set-associative mapping and address permutation, single-bit correcting and double-bit detecting codes (SEC-DEC codes) are used for both reliability and yield improvement. Fugiwara and Tanaka used a combination of fault tolerance techniques to achieve higher yields, a phenomenon investigated earlier by Stapper in [65]. They also indicate that their large file memory system could fill the speed-gap between main memories and disk [21].

3.2.5 Error-Correcting Codes and Redundancy

As mentioned in Chapter 2, using a co-ordinated combination of ECC and redundancy schemes provides far greater fault tolerance than either method used alone. These various schemes can be quantified and compared using yield and fault models. A binomial yield model will be used in conjunction with fault models to predict yields for different redundancy schemes. That is, various combinations of ECC, row and column redundancy have different effects on overall DRAM yield. The yield models were presented for a 16-Mb DRAM by Stapper in [65] and generalized by Wickman [81] and Joly [33].

DRAM fabricators improve yield using various schemes that increase the error tolerance of a device. Row redundancy involves the use of spare rows that can replace non-functional wordlines or rows with multiple defects. Likewise, column redundancy replaces faulty columns with spares. Error-correcting codes (ECC) commonly use modified Hamming codes to correct or detect DRAM bit errors. Check bits are created from a set of data bits and are stored with these data bits in

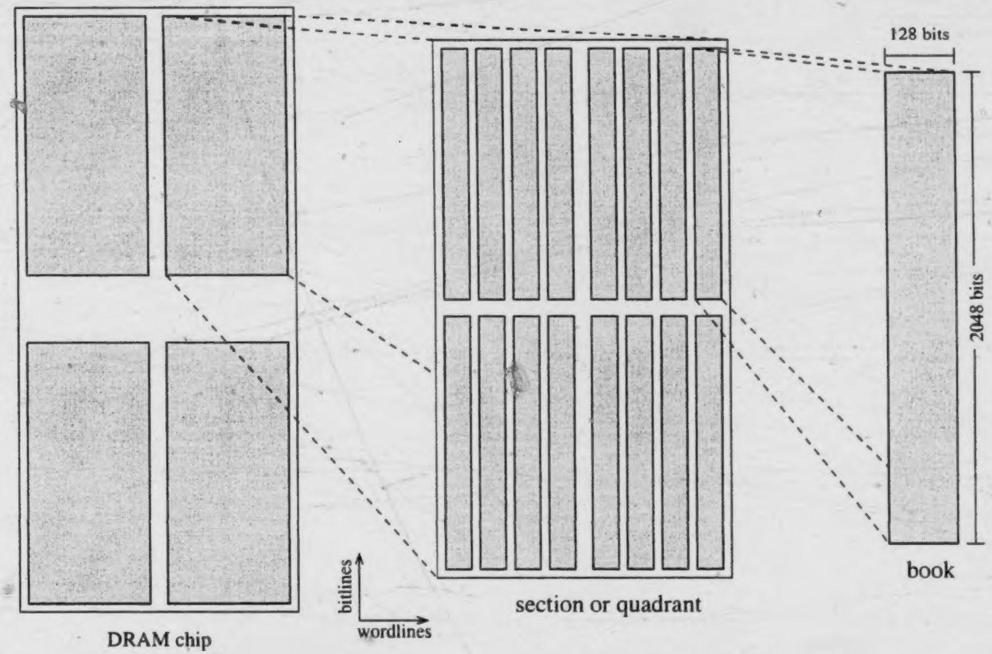


Figure 3.2: Simplified block diagram of the IBM 16-Mb DRAM. Redundancy is omitted for simplicity. Although shown as discrete blocks, books are interweaved along the word lines [33, 65].

memory to form a code word. When the data bits are read, the check bits are recalculated and compared against the original check bits. The result of the comparison indicates if an error has occurred [82]. It is possible that an error-correcting code can correct a single error but detect two erroneous bits. This type of ECC provides for single-error correction and double-error detection (SEC-DED) [65].

The yield models that will be discussed shortly are based on the organization of an IBM 16-Mb DRAM chip [34, 65, 66]. As outlined in Figure 3.2 and Table 3.2, this chip is divided into four quadrants or sections. A *section* is the area of memory to which row redundancy is applied. Ignoring redundancy and ECC, each section of the 16-Mb chip has 4096 wordlines and 1024 bitlines. The redundant wordlines are associated with each section while ECC and column redundancy are applied to a *book*. Since each section uses 72 extra bitlines to support ECC, there are actually 1096 bitlines in a section. A set of 128 bits along a wordline is associated with an

Table 3.2: IBM 16-Mb DRAM and Samsung 1-Gb DRAM Specifications [44, 65]

	16-Mb DRAM	**1-Gb DRAM**
Nominal capacity	16 Mb	1 Gb
Banks	1	4
Sections per bank	4	2
Books per section	16	4
Pages per book (wordlines)	2048	4096
Bits per page (bitlines)	128	8192
Redundant rows per section	24	64
Redundant columns per book	2	16
Bits per ECC code word	137	523
ECC code words per book	2048	65536

additional 9 bits, forming a 137-bit ECC code word. That is, an *ECC code word* is the complete ECC word that includes both data bits and check bits. The books and ECC code words actually are interweaved along the wordline of a section so that paired cell failures along the wordline can be corrected by different ECC code words [65].

Yields for the 16-Mb chip will be compared against those for the Samsung 1-Gb DRAM chip with a similar chip organization [33, 44]. Specifically, yields for the 16-Mb and 1-Gb DRAMs with various combinations of redundancy and ECC are shown in Figures 3.3 and 3.4, respectively. These yields are expressed in terms of the average number of faults per memory cell. Table 3.3 compares the number of tolerable faults per cell at 50% yields for the various configurations.

3.2.5.1 No Redundancy or ECC

The simplest yield model involves a DRAM with no redundancy or ECC. Assuming a Poisson distribution of faults, the yield of a single DRAM cell is defined by

$$Y_{sc} = e^{-\lambda_{sc}} \qquad (3.1)$$

Table 3.3: The Tolerable Faults per Die at 50% Yield for Two DRAMs

Redundancy Scheme	**16-Mb DRAM (max faults/die)**	**1-Gb DRAM (max faults/die)**
No ECC or Redundancy	< 1	< 1
Row and Column Redundancy	240	1400
ECC but No Redundancy	400	1675
ECC, Row and Column Redundancy	7500	76000

where λ_{sc} represents the average number of faults per cell [65]. The yield of a DRAM cell is the probability a given cell will be functional. The yield of a DRAM device, Y_{DRAM}, is then

$$Y_{DRAM} = Y_{sc}^{b_{DRAM}} \tag{3.2}$$

where b_{DRAM} is the number of bits in the DRAM. From Equation (3.2), there must be less than one fault per DRAM device to achieve better that 50% yield. This is an extremely small fault density and demonstrates why redundancy is essential [33]. The yield for a 16-Mb DRAM is shown in Figure 3.3 while the yield for the 1-Gb DRAM is so low that it is not visible in Figure 3.4.

3.2.5.2 Row and Column Redundancy

The yield of a column of DRAM is given by

$$Y_{col_{RC}} = Y_{sc}^{b_c} \tag{3.3}$$

where b_c is the number of bits in a column. Using this yield, the yield of a book is calculated by summing all contributions to the yield distributions from zero errors to the number of redundant columns

$$Y_{book_{RC}} = \sum_{i=0}^{r_c} \binom{n_c + r_c}{i} \cdot Y_{col_{RC}}^{n_c+r_c-i} \cdot \left(1 - Y_{col_{RC}}\right)^i \tag{3.4}$$

where n_c is the number of columns in the book (page size) and r_c is the number of redundant columns per book. The first term of Equation 3.4 "chooses" which columns are bad, the second is the yield of the functional columns and the third is the yield of the faulty columns.

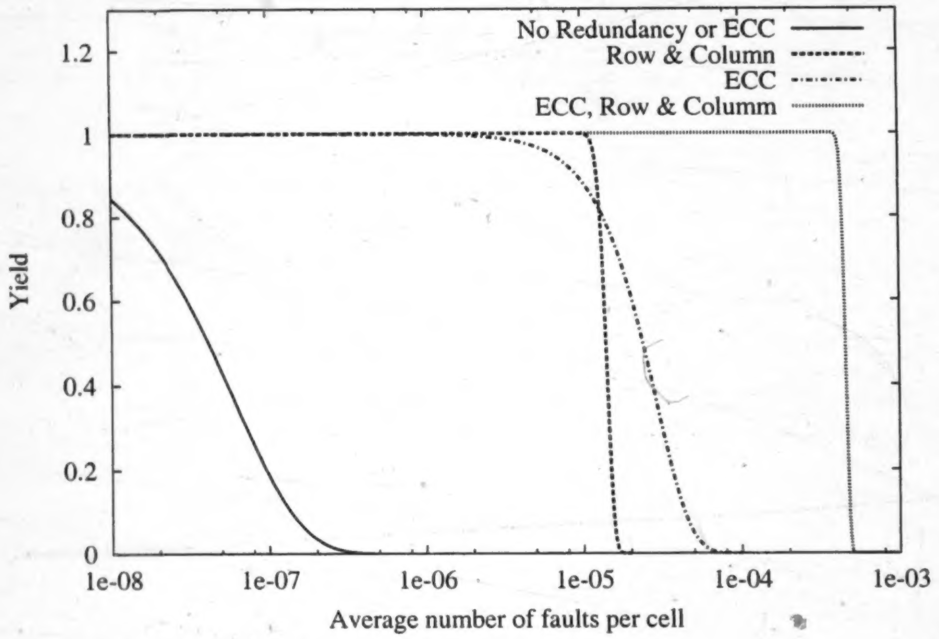


Figure 3.3: Yields for the IBM 16-Mb DRAM.

The yield of a row for a DRAM with column redundancy is

$$Y_{row_{RC}} = Y_{book_{RC}}^{\frac{b_r}{b_b}} \tag{3.5}$$

where b_r is the number of bits in a row and b_b is the number of bits in a book. Equation (3.5) uses an estimate of the effective yield of a single cell inside a book as the yield for each of the bits of a row. The yield of a section now can be calculated

$$Y_{sec_{RC}} = \sum_{i=0}^{r_r} \binom{n_r + r_r}{i} Y_{row_{RC}}^{n_r+r_r-i} \left(1 - Y_{row_{RC}}\right)^i \tag{3.6}$$

where n_r is the number of rows in a section and r_r is the number of redundant rows per section. As graphed in Figures 3.3 and 3.4, the yield of the DRAM with redundant rows and columns is then given by

$$Y_{DRAM_{RC}} = Y_{sec_{RC}}^{n_s} \tag{3.7}$$

where n_s is the number of sections in the DRAM.

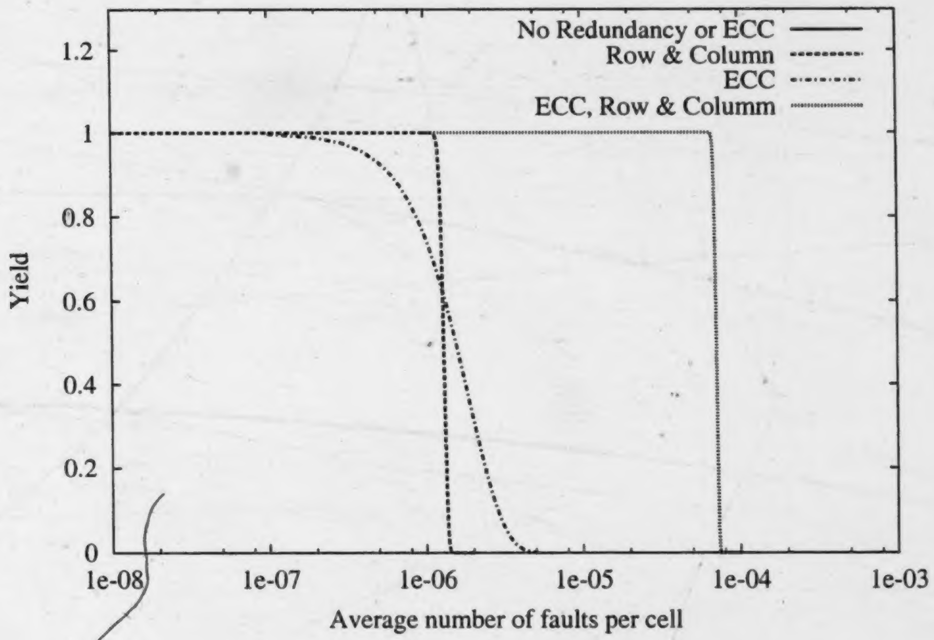


Figure 3.4: Yields for the Samsung 1-Gb DRAM.

3.2.5.3 ECC

The next step is to determine an equation for the yield of a DRAM with ECC. The yield of a single-error correcting (SEC) ECC code word can be calculated as

$$Y_{cw} = Y_{sc}^{b_{cw}} + \binom{b_{cw}}{1} Y_{sc}^{b_{cw}-1} \left(1 - Y_{sc}\right) \tag{3.8}$$

where b_{cw} with the number of bits in the code word. The yield of a book in a DRAM with ECC is calculated as

$$Y_{book_E} = Y_{cw}^{n_{cw}} \tag{3.9}$$

where n_{cw} is the number of code words in a book. The yield of a DRAM with ECC is then

$$Y_{DRAM_E} = Y_{book_E}^{n_b} \tag{3.10}$$

where n_b is the number of books in the DRAM. Graphs of Equation (3.10) for the two DRAM chips are presented in Figures 3.3 and 3.4.

3.2.5.4 Row and Column Redundancy plus ECC

It now is possible to determine the effective yield of a DRAM with ECC as well as row and column redundancy. The effective yield of a single cell of a DRAM with ECC depends on the yield of a code word found using Equation (3.8). Therefore, the effective yield of a column is

$$Y_{col_{RCE}} = Y_{cw}^{\frac{b_c}{b_{cw}}} \tag{3.11}$$

where b_c is the number of bits in a column. The effective yield of a book with ECC and column redundancy is

$$Y_{book_{RCE}} = \sum_{i=0}^{r_c} \binom{n_c + r_c}{i} Y_{col_{RCE}}^{n_c+r_c-i} \left(1 - Y_{col_{RCE}}\right)^i . \tag{3.12}$$

Using the effective yield of a book and the effective yield of a single cell in this book, the effective yield of a row of a DRAM with ECC and column redundancy is

$$Y_{row_{RCE}} = Y_{book_{RCE}}^{\frac{b_r}{b_b}} \tag{3.13}$$

which is similar to Equation (3.5). Now row redundancy can be considered. The yield of a section with ECC, column and row redundancy is

$$Y_{sec_{RCE}} = \sum_{i=0}^{r_r} \binom{n_r + r_r}{i} Y_{row_{RCE}}^{n_r+r_r-i} \left(1 - Y_{row_{RCE}}\right)^i \tag{3.14}$$

which has the same form as Equation (3.6). Finally, the yield of a DRAM with ECC, redundant rows and redundant columns is given by

$$Y_{DRAM_{RCE}} = Y_{sec_{RCE}}^{n_s} . \tag{3.15}$$

As discussed in [65], combining ECC with row and column redundancy has a synergistic effect on DRAM yields. According to Equation (3.15), the 16-Mb DRAM is able to achieve a 50% yield ratio with up to about 7500 faults (see Table 3.3). The 1-Gb DRAM can tolerate about 76000 faults at 50% yield. As shown in Figures 3.3 and 3.4, the yields for these configurations are substantially better than the other DRAM yields presented earlier. Using redundancy in combination with ECC produced much better fault tolerance than the sum of the redundancy of either method functioning alone. This synergism can be explained by a complementary effect between ECC and redundancy: Clusters of faulty cells can be broken up so that the ECC mechanism is able to correct single-bit errors. Therefore, DRAM chips with ECC and redundancy have better yield for a lower overall cost per bit [82]. However, yields can be improved further if the application does not require a contiguous address space and can map around bad blocks.

3.3 Device-Level Bad Block Marking

In [81], Wickman proposed that DRAMs incorporate a mechanism for identifying faulty blocks, which would improve yield and permit the earlier introduction of future generation DRAM chips. The yield of partially-good memory products can be calculated to help predict the yields of file memories that would be shipped with known faulty blocks. That is, chips with small numbers of defects could be marketed as file memories instead of being discarded. The proposed device-level design involves enhancing DRAM chips with small circuits that mark bad blocks using data structures in non-volatile memory. The focus of this effort was to utilize as much of the functional memory cells as possible.

3.3.1 Partially-Good Product

In some cases, failures occur in a particular section of a chip while the remainder of the chip is defect-free. For example, one quarter of the chip may be unusable while the rest of the chip is functional. If the sections with faults are disabled, then a partially-good chip is available to be packaged with similar partially-good chips in a memory module that is marketed with the capacity as the sum of the functional sections. By recognizing the fact that defects tend to cluster [7], this discussion will expand on the concept of partially-good product presented in [67].

In [67], the equivalent yield of partially-good product is defined as the fraction of usable capacity and is given by

$$Y_{EQ} = Y_{AG} + \frac{k}{n} Y_{PG} \tag{3.16}$$

where Y_{AG} is the all good yield and (k/n) is the fraction (i.e. 1/2, 3/4, ...) of usable capacity for partially good chips. If file memory can be subdivided into n blocks, it is useful to be able to determine the probability that k of the blocks are good. According to [67], this probability is

$$Y_{PG} = \binom{n}{k} Y_0 Y_K Y_b^k (1 - Y_b)^{n-k} \tag{3.17}$$

where Y_0 is the gross yield, Y_K is the probability that a chip contains no killer defects, and Y_b is the yield of each block. Using the binomial theorem,

$$(a+b)^n = \sum_{j=0}^{n} \binom{n}{j} a^{n-j} b^j \tag{3.18}$$

Equation (3.17) can be written as

$$Y_{PG} = \binom{n}{k} Y_0 Y_K \sum_{j=0}^{n-k} \binom{n-k}{j} (-1)^j \, Y_b{}^{j+k} \tag{3.19}$$

according to [67].

The yields in Equation (3.19) could be modeled with the yield equations discussed in Section 3.2.5. However, modeling yields such as Y_K with Poisson yield equations is pessimistic since defects are not randomly distributed, but tend to cluster [7]. Instead, it should be assumed that the defects are distributed according to the negative binomial distribution to account for the clustering of defects. Then the probability that a chip contains no killer defects is given by

$$Y_K = \left(1 + \frac{\lambda_K}{\alpha}\right)^{-\alpha} \tag{3.20}$$

where λ_K is the average number of killer defects and α is the defect clustering parameter, whose values typically range from 0.5 to 5 for different fabrication processes [6]. The yield of a chip after repair of defects, Y_{Keff}, is the same as (3.20), except that λ_K is reduced to λ_{Keff}. That is, if p_R is the probability that a defect can be repaired, then $\lambda_{Keff} = (1 - p_R)\lambda_K$. The probability of m killer defects is

$$P[K(m)] = \frac{\Gamma(\alpha+m)}{m!\,\Gamma(\alpha)} \frac{\left(\frac{\lambda_K}{\alpha}\right)^m}{\left(1 + \frac{\lambda_K}{\alpha}\right)^{\alpha+m}} \tag{3.21}$$

where $\Gamma(x)$ is the gamma function [6]. This permits calculation of the fraction of functional chips with exactly m repairs:

$$f(m) = \frac{p_R^m P[K(m)]}{Y_{Keff}}. \tag{3.22}$$

As discussed later in this chapter, the distribution of repairs is the distribution of faults that need to be marked as bad.

This analysis has ignored the issue of latent defects, which escape wafer probe testing and cause early-life reliability failures. A repaired die has a greater chance of containing latent defects since defects tend to cluster. However, the average number of latent defects is 1% – 2% of the average number of killer defects. A more detailed discussion of reliability yield can be found in [6] and [7].

Using partially good memories is similar to *performance binning*, which involves separating integrated circuits into bins based on operating frequency. Instead, creating different classes of partially good chips offer the ability to improve product yield [61]. Designs may contain non-essential components that are solely intended to improve system performance. When a defect is detected within a component, it is disabled with a reduction in performance.

3.3.2 Methods and Results

Wickman discussed a variety of methods of bad block marking that require a small quantity of non-volatile memory [81]. The first method uses a block bitmap where each bit in the bitmap represents a block and a bit is set if a block contains one or more defects. Reducing the block size increases the number of functional bits that are recoverable but it also increases the size of the bitmap. Another data structure for marking bad blocks is list marking, which lists the starting address of each block. The most appropriate bad block marking method depends on the number of non-volatile memory bits and the number of bad blocks. Bitmap marking is used instead of list marking unless the number of bits that are required to perform list marking is less than than the number of non-volatile bits. For the case where the number of defects is very large, list marking could be used to mark the remaining good blocks.

Bad block marking is able to substantially improve chip yield. For the 1-Gb DRAM, Wickman found that 89559 defects can be tolerated at a 50% equivalent yield when ECC, row and column redundancy and bad block marking are combined. When only bad block marking is used, 90705 defects can be tolerated due to the absence of the overhead required for redundancy.[2] The mask layout area overhead of the non-volatile memory for bad block marking was 0.12% while the redundancy mechanism adds 0.82% in overhead.

3.3.3 Discussion

Wickman's implementation of bad block marking is based on several design decisions that create unnecessary performance penalties and increase fabrication costs. First, Wickman's various methods for marking bad blocks have a negative impact on memory performance since the data structures need to be searched for every memory access. The impact on the performance of sequential accesses using a simulated DRAM with bad block marking was 75% that of conventional DRAM. Second, the bad block marking algorithm relied heavily on sequential accesses to file memory. While sequential accesses are common within pages, a disk cache backing large numbers of small files will have a higher frequency of random accesses.[3] Third, the defect list or bitmap requires some form of non-volatile memory which introduces additional hardware cost or process technology steps. Each of these issues are addressed in a new method for marking bad blocks at memory allocation rather than during every memory access.

3.4 System-Level Bad Block Marking

Design of a file memory device exclusively at the level of the device ignores issues that are more effectively handled by the operating system. For example, a storage medium with noncontiguous allocatable regions requires algorithms for managing data in ways that reduce fragmentation. One such algorithm is the (binary) buddy system that is commonly used for dynamic memory allocations. In the proposed method, this buddy system algorithm is adapted so that it is able to improve the fault tolerance of extended storage. That is, the operating system's existing buddy system implementation can mark bad blocks without adding additional overhead to memory access operations. Fine-grained bad block marking is possible using various techniques that recover the functional portions of a page for further yield improvements.

---

[2] Wickman's results are based on different assumptions regarding the organization of the DRAMs. For example, the 50% yield point for the 1-Gb was 54751 faults instead of 76000 faults.

[3] Since performance was drastically reduced for random access tests, Wickman used a binary tree data structure instead of a linear data structure. The performance of this scheme was 60% that of conventional DRAM.

3.4.1 Advantages

A system-level approach has a number of advantages over marking bad blocks at the level of the DRAM device. This involves modifying the physical memory allocator of the operating system so that faulty blocks within the physical ESDC address space are never allocated. The main advantage of this approach is that there is virtually no overhead—a data structure representing bad blocks does not need to be searched before every memory access. Instead, the mapping of bad blocks is initialized during operating system initialization. As well, the method permits variable bad block sizes that are integer powers of two. With this method, nonsequential accesses to different blocks of extended storage will not adversely affect performance. A portion of non-volatile memory is unnecessary since the map of bad blocks is shipped as part of an extended storage device driver. Finally, if lists of the faults for current downgraded DRAMs are made available, extended storage would be possible without hardware changes.

3.4.2 Binary Buddy System for Memory Allocation

Several operating systems manage and allocate physical pages of memory using the *Binary Buddy System*. The buddy system was first described in 1965 by Knowlton [41] and later by Knuth [42]. It is a fast dynamic memory allocation algorithm that is designed to handle external fragmentation. That is, it avoids the situation where it is not possible to allocate a large block of contiguous page frames even though enough non-contiguous free page frames are available. Memory is dynamically allocated by the buddy system for use by processes, for certain kernel data structures, for virtual disk caches, and for other purposes.

In the buddy system, blocks must be powers of two in size and certain adjacent free blocks can be coalesced. In Linux, memory is divided into blocks of page frames, while block sizes smaller than a page are possible in operating systems such as BSD Unix [20]. When the algorithm is asked to allocate a block of memory and if the desired size is not available, a larger block is split in half. The two resulting smaller blocks are *buddies*. One buddy block is free while the other block is used for the allocation. The latter block may be sub-divided further until a block of suitable size is made available. Later, when two buddies are available, they are coalesced into a single block.

3.4.2.1 Data Structures

In Linux, the buddy system uses the data structures shown in Figure 3.5. Physical memory is subdivided into *page frames*, which are also known as physical pages. *Page descriptors* are the kernel data structures that keep track of the status of page frames. They are less than 64 bytes in sizes and are stored in an array, as shown in Figure 3.5. A "free area" array maintains ten doubly-linked circular lists of free blocks of page descriptors. The index k of this array is the order of the number of blocks of page frames. Each k^{th} element of this array points to a bitmap, which keeps track of buddy blocks of size 2^k page frames. A pair of buddy blocks may be free or busy if the bit of the bitmap is set to 0. However, exactly one of the buddies is busy if the bit is 1. The size of the bitmap for array element k is half the size of the bitmap for array element $k-1$.

The buddy system is a very efficient memory allocator; if the address and size of a block is known, then the address of the buddy block can be found very quickly [42]. The address of a block of size 2^k is always a multiple of 2^k. Therefore, if a block of size 2^k is located at address a, then the address of the buddy is a XOR 2^k.

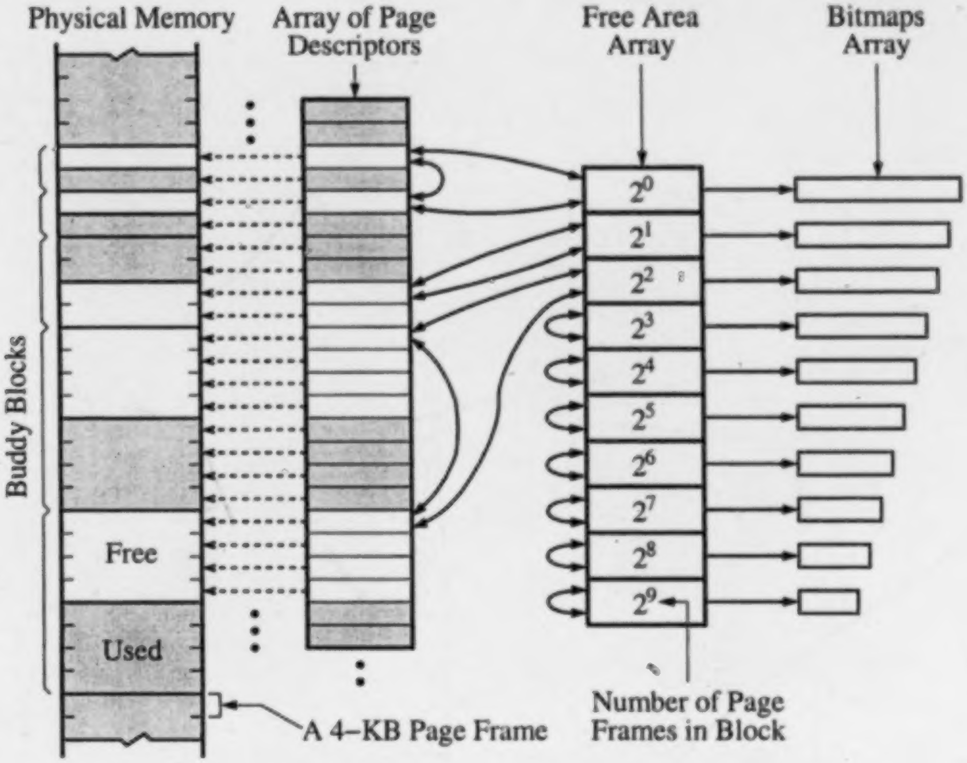

Physical Memory
Array of Page Descriptors
Free Area Array
Bitmaps Array
Buddy Blocks
Free
Used
2^0
2^1
2^2
2^3
2^4
2^5
2^6
2^7
2^8
2^9
A 4–KB Page Frame
Number of Page Frames in Block

Figure 3.5: Buddy system data structures [8, p. 236].

This is useful both for allocating blocks and for merging free buddy blocks into larger blocks.

3.4.2.2 Block Allocation

Sometimes the buddy system is requested to allocate a block but no blocks of the requested size are available. Instead, a block of a larger size 2^k is selected and split in half to allocate the block with the desired size of 2^i. Two buddy blocks of size 2^{k-1} result from splitting the 2^k block. If one of these buddies is still too large, then it is split in half. This process of splitting blocks is repeated until a set of blocks of sizes $2^{k-1}, 2^{k-2}, \ldots, 2^{i+1}, 2^i, 2^i$ is obtained. All blocks in this set are still free blocks except for one of the blocks of size 2^i. It is marked as "allocated" using the associated bitmap and is removed from its free area list [20].

3.4.2.3 Block Reclamation

Whenever a block is freed, the buddy system checks the associated bitmap to determine if this block can be be merged with its buddy block. If this is possible, a free block twice the size as the original block is created. The algorithm repeats the process for the larger block, checking if it can be merged with its buddy. When individual pages are freed in Linux, block merging is repeated at most nine times. Coalescing free blocks reduces external fragmentation, which improves the success of subsequent large memory allocations.

3.4.3 Binary Buddy System for Fault-Tolerance

While studying Knowlton's explanation of the buddy system, I discovered that this algorithm is not just applicable to fast dynamic memory allocation. A closer examination of the representation of physical memory in Figure 3.5 revealed that the buddy system could be enhanced to improve the fault-tolerance of file memory when it is employed as extended storage. File memory devices would continue to benefit from the standard yield enhancement techniques of ECC and redundancy. However, the yield would be improved further by using the buddy system to ensure that bad blocks are never accessed by the operating system as extended storage or for any other purpose. The main advantage of this method is that blocks are marked as bad during operating system initialization rather than during every memory access. The fault-tolerant buddy system may be implemented entirely by the operating system for page-level bad block marking. If extended storage hardware does not enforce page addressability, finer bad block resolution that wastes fewer good bits is possible. While the existing buddy system handles allocations based on pages, a separate buddy system could manage blocks smaller than a page specifically for fault tolerance. If extended storage must be page-addressable, sub-page bad block marking could be implemented in hardware instead of at the operating system domain. It should be mentioned that other memory allocators were not considered for fault tolerance since the buddy system is already implemented in the kernel.

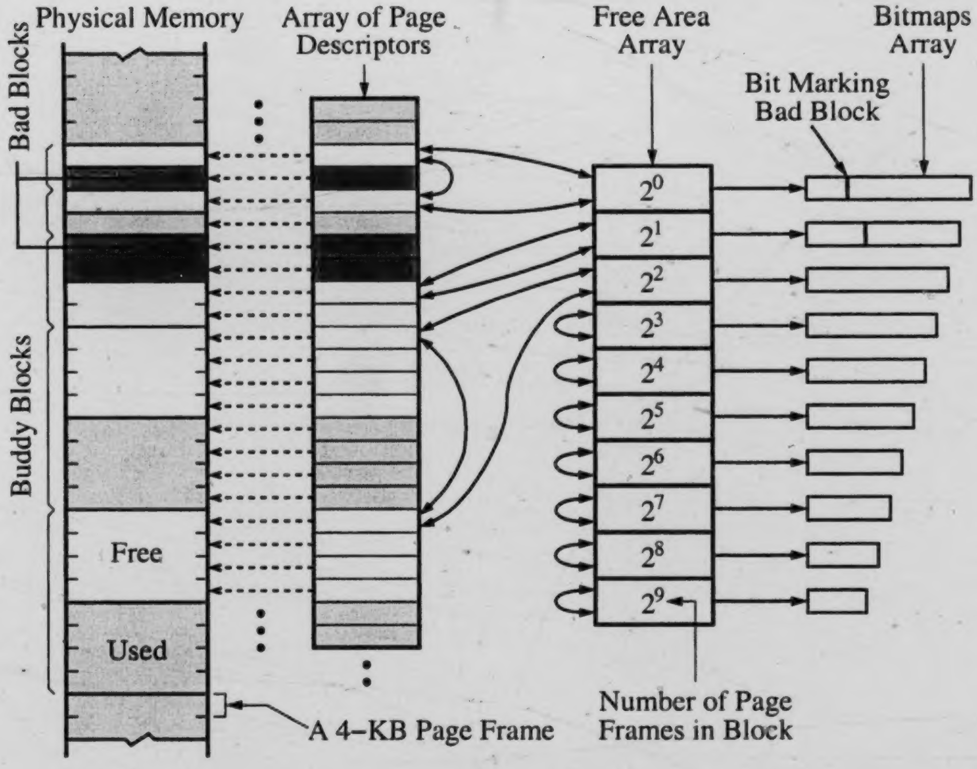

Bad Blocks
Physical Memory
Array of Page Descriptors
Free Area Array
Bitmaps Array
Bit Marking Bad Block
Buddy Blocks
Free
Used
2^0
2^1
2^2
2^3
2^4
2^5
2^6
2^7
2^8
2^9
A 4–KB Page Frame
Number of Page Frames in Block

Figure 3.6: Bad block marking by block allocation feigning does not change buddy system data structures.

3.4.3.1 Page-Level Bad Block Marking

With minor operating system kernel modifications, the dynamic memory allocator can be exploited to manage bad blocks that are multiples of page frames in size. The buddy system already identifies allocated blocks and free blocks, as shown in Figure 3.5. If a set of the physical addresses of faulty memory cells is known, the pages or blocks where these faults occur can be marked. For a given block size of 2^k, the block address that contains a fault can be obtained efficiently:

$$addr_b = -2^k \text{ AND } addr_f \tag{3.23}$$

where $addr_f$ is the address of a faulty word in memory. Now the task is to determine an appropriate method to mark such faulty blocks as unusable.

A novel bad block marking method has been devised, which involves feigned allocations of faulty blocks during operating system initialization. In a *feigned allocation*, the page frames associated with the faulty blocks are allocated but never are used for actual data. This is shown in Figure 3.6, which is identical to Figure 3.5 since no new data structures are added. Therefore, it is possible to feign a set of dynamic memory allocations that can mark all bad blocks of 2^k page frames in size.

There are two techniques of implementing the feigned allocation technique in Linux. First, an existing kernel daemon could be modified to allocate all of the faulty blocks so that they can never be utilized. These allocated blocks must be configured so that it is impossible to free them or to swap them to disk. The advantage of this technique is that the buddy system implementation is not changed. However, the affected kernel daemon has been burdened with an additional responsibility and such an implementation could have unforeseen consequences unless carefully designed. A more subversive technique accomplishes the same effect as daemon-based allocation feigning by directly modifying the buddy system data structures. Instead of using a kernel daemon, bad blocks are marked as allocated during data structure initialization. The overhead of either technique is negligible since it is incurred only during system initialization. The feigned allocations are permanent since they can not be paged to backing store or freed by page reclamation mechanisms.

An alternative method extends the data structures used by the buddy system in Linux. This method adds a second set of bitmaps that are referenced by elements of the *free area array*, where the bitmaps are initialized with the location of bad blocks in memory. For every allocation request, the buddy system consults these bitmaps to determine if a candidate block should be avoided since it is bad. This alternative method introduces more overhead than the first method. Since additional bitmaps are consulted for every memory allocation, the performance of dynamic memory allocation operations will be affected. However, bad block checking at every allocation attempt will still offer superior performance results than checking for bad blocks before each access of a memory word.

Adding a complete set of bitmaps to the buddy system data structures can significantly increase static memory requirements for systems with large memory capacities. For this reason, the memory footprint of the buddy system bitmaps needs to be quantified. The number of bytes in a bitmap associated with a group of blocks of a given order k is

$$s_k = \frac{1}{8}\frac{s_m}{2^{k+1}s_p} \tag{3.24}$$

where s_m is the size of the memory managed by the buddy system and s_p is the size of a page frame.[4] The order is increased by one since each bit of a bitmap manages a pair of buddy blocks. The size of a complete set of bitmaps in bytes is given by

$$s_{set} = \frac{s_m}{8s_p}\sum_{k=0}^{k_{max}-1}\frac{1}{2^{k+1}} \tag{3.25}$$

where k_{max} is the maximum order of the buddy system. In Linux, k_{max} is 10 since there are ten possible block sizes, ranging from a single page frame to 512 page frames. For example, a gigabyte of extended storage requires 32736 bytes (just less than 32 KB) for a complete set of bitmaps, according to Equation (3.25). Thus, duplicating the entire set of bitmaps for marking bad blocks would increase the memory requirements of the operating system. Therefore, any method of page level bad block marking should avoid duplicating a complete set of bitmaps.

3.4.3.2 Sub-Page-Level Bad Block Marking

Page-level bad block marking can be accomplished with virtually no overhead in terms of memory requirements or reduced system performance. However, the argument can be made that entire page frames become unusable even in the presence of a single faulty bit. Therefore, if faulty blocks smaller than a page frame could be avoided, the cost-effectiveness of extended storage memory could be further improved. This thesis proposes a number of designs for sub-page-level bad block marking. First, single pages could be stored in extended storage across several faulty page frames. Second, real-time compression algorithms could be used to store all 4096 bytes of a page in functional portions of faulty page frames. However, if extended storage is only page-addressable due to device addressing limitations, sub-page-level bad block marking in hardware must be considered.

---

[4] As will be discussed in detail in Chapter 4, Linux uses a different buddy system for each zone of memory for Intel architectures. Since ESDC is restricted to a single zone, the memory managed by the buddy system for this zone is equivalent to total ESDC memory capacity.

If a page frame contains one or more faults, the majority of the bits in the page frame likely are still functional. Pages of data could be efficiently stored across several faulty page frames if the faulty portions are known. A buddy system implementation could be used to manage functional sub-blocks that are powers of two in size, but are smaller than the 4096-byte page frame. This buddy system would be distinct from the buddy systems used for dynamic memory allocation. It only would be needed to manage good sub-blocks within faulty pages. Good sub-blocks would range in size from one word (4 bytes) to half a page (2048 bytes).

A sub-page-level buddy system has performance advantages but consumes OS memory resources. First, the sub-page components of the buddy system algorithm are executed only during extended storage page allocations instead of during each access to extended storage. Second, this mechanism only is utilized for pages with faulty sub-blocks. Nevertheless, this method requires additional operating system memory resources. A new bitmap is required to indicate which pages require sub-page-level bad block marking. This bitmap only manages blocks that are the size of page frames, so it would occupy 16 KB by Equation (3.24). A doubly-linked list of page descriptors corresponds to the pages with faulty sub-blocks. Each of those pages requires a set of buddy system bitmaps; from Equation (3.25), such bitmaps would require up to 256 bytes per page. Therefore, other solutions, possibly involving a hardware implementation, need to be investigated to reduce sub-page-level bad block marking overhead.

3.4.4 Feasibility

A system-level bad block marking design has to be feasible if it is to be commercially successful. A number of different factors affect the feasibility of this design. First, the scope of the operating system enhancements and possible hardware modifications will influence the design's marketability. Second, the design's feasibility

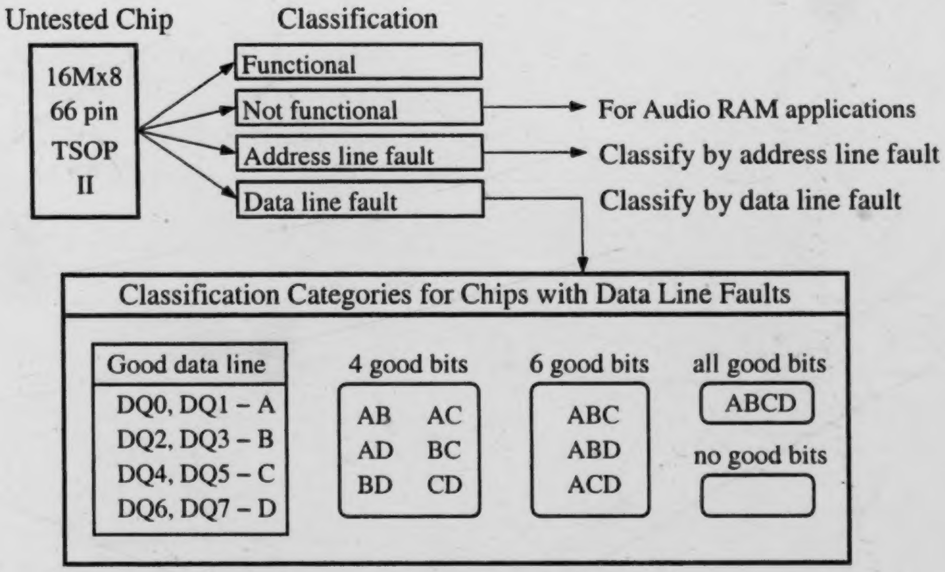

Untested Chip
Classification
16Mx8
66 pin
TSOP
II
Functional
Not functional
Address line fault
Data line fault
For Audio RAM applications
Classify by address line fault
Classify by data line fault
Classification Categories for Chips with Data Line Faults
Good data line
DQ0, DQ1 – A
DQ2, DQ3 – B
DQ4, DQ5 – C
DQ6, DQ7 – D
4 good bits
AB AC
AD BC
BD CD
6 good bits
ABC
ABD
ACD
all good bits
ABCD
no good bits

Figure 3.7: Downgraded DRAM classification process [27].

will be determined by the extent of how memory manufacturers manage defective memory devices. Finally, reliability issues have a direct impact on overall feasibility of the concept.

Operating system enhancements are necessary for a system-level bad block marking design while hardware changes are optional. OS modifications should be minimized in terms of their impact to ensure stability and portability. Conceptually, the design would be released as a Linux kernel patch for verification by the kernel development community. A robust design would reduce the time required for the patch to appear in an official kernel release. If hardware changes are necessary to improve cost-effectiveness of extended storage, it may be difficult to convince DRAM fabricators to incorporate the design into a proven product line. Therefore, the hardware design component may be more appropriately introduced at the level of multi-chip memory modules, such as single in-line memory modules (SIMMs).

DRAM manufacturers do not discard chips that fail final functional testing stages since the chips are already packaged. As outlined in Section 1.4.3, such downgraded DRAMs are sold as “C grade” chips for a variety of applications [27].

These chips are sorted into categories (bins) by types of defect patterns, as shown in Figure 3.7. Downgraded chips can be integrated to produce a functional memory module, even though extra chips are required to account for the faulty portions.[5] However, such defect patterns are particularly suitable for page-level bad block marking. Large blocks are known to be faulty while other areas of the chip are functional. A large-capacity extended storage memory module could be constructed if multiple partially-good DRAMs are assembled onto a memory module. Sub-page level bad block marking would be necessary to handle downgraded DRAMs that currently are useful only for audio applications. Therefore, extended storage is feasible using available downgraded memory in conjunction with operating system enhancements, such as those discussed in this thesis.

System-level bad block marking also depends on the availability of information regarding faulty memory cells or blocks. Such information can only be obtained during testing by chip fabricators or memory module manufacturers. Post-consumer memory testing is not practical due to the requirement for burn-in testing at an average of 125 degrees Celsius, large voltage ranges, and sensitive DC current measurements [3, 27]. Unlike faults in conventional DRAMs, a list of faults for a downgraded chip or memory module must be provided to an extended storage device driver. A manufacturer that would market large volumes of downgraded DRAM must manage this data regarding the location of faults for every chip. Since each product has a unique pattern of defects, a serial number on each product could be used to access downloadable fault information stored in a corporate database. However, this approach could require significant resources, as hypothetically demonstrated in Table 3.4. To reduce costs, fault information should be included with the product as a device driver.

Downgraded DRAMs are considered to be substandard due to compromises on performance, durability, longevity, or data retention. Therefore, consumers would be reluctant to use such DRAMs for main memory. However, if downgraded DRAMs are sufficiently tested to determine the locations of faults, they would be

---

[5] It is not clear why the author of [27] did not include *BCD* as a classification category.

Table 3.4: Estimates of Worst-case Resources Required for Maintaining a Downloadable Bad Block Marking Database for Every Module Sold

Parameter	**Estimate**
Number of faults per 256-Mb chip	100000
Size of address of faults	4 bytes
Uncompressed fault list per chip	400 KB
Compressed fault list per chip	200 KB
Compressed fault list per 16-chip SIMM	3.2 MB
Number of modules sold per year	250 million
Database size to track sold modules	800 TB

5

immediately marketable as extended storage. This is possible since some methods of system-level bad block marking only require operating system enhancements rather than DRAM design modifications. However, even if DRAMs pass thorough testing, transient faults may still appear. Therefore, it also is essential that extended storage uses DRAMs with ECC so that transient errors can still be repaired.

3.5 Conclusion

Techniques for repairing or avoiding bad blocks have been developed for disks, caches and flash memories. Previous work has discovered that combining redundancy, error correction codes and bad block marking produces far greater fault tolerance than the sum of any of the methods functioning alone. Yields for partially good products have also been investigated in previous work, but recognizing that defects tend to cluster is important for avoiding pessimistic yields.

A system-level approach to bad block marking for extended storage primarily involves operating system enhancements. Hardware modifications may be required to improve efficiency or yield, but an independent hardware design can not exploit the benefits of marking bad blocks as feigned memory allocations. Just as memory is best managed by an operating system [25, p. 394], bad block marking is most efficiently handled by the operating system for blocks larger than a page.

Bad block marking using the buddy system creates discontinuities in physical memory. In operating systems, sometimes contiguous physical page frames are necessary [8, p. 233]. For example, DMA ignores the paging circuitry and accesses the address bus directly, so the affected memory must be an area of contiguous physical memory. However, discontinuities are possible within the physical address space belonging to extended storage. As will be explained in detail in Chapter 4, DMA transfers can never directly access file memory, so ESDC is compatible with non-contiguous memory allocation.

> ... the smaller blocks are obtained by successively splitting larger ones in half, and the larger blocks are reconstituted if and when their parts are simultaneously free.
>
> — Kenneth C. Knowlton, 1965
> *Describing a fast storage allocator,*
> *now known as the buddy system* [41]

Chapter 4

Extended Storage Disk Cache

4.1 Introduction

File memory has a lower cost per bit than conventional DRAM but has reduced performance, due to slower or partially-good product. It is possible that file memory can improve the performance of memory-based systems and devices when used as extended storage, but the quantity and performance of file memory must be determined for extended storage to be an economical solution. Since file memory also is faster than disk media, it would be an excellent candidate for solving the growing performance gap between disk media and conventional DRAM. Thus, a new stage is proposed to appear in the memory hierarchy of a computer system. In this design, extended storage would serve as an extended storage disk cache (ESDC).

4.2 ESDC Design Overview

ESDC is designed to be a general-purpose stage in the memory hierarchy, a disk cache that functions as both a file I/O cache and a virtual memory cache. That is, ESDC is a cache of pages associated with files on disk as well as a cache for virtual memory swap space. As shown in Figure 4.1, a conventional memory hierarchy often employs portions of main memory not allocated to processes to function as a dynamic virtual disk cache for disk files and swap space [8]. Such an approach is vulnerable to reduced disk cache sizes during periods of intensive memory usage. In the ESDC architecture, this cache is moved to extended storage that consists

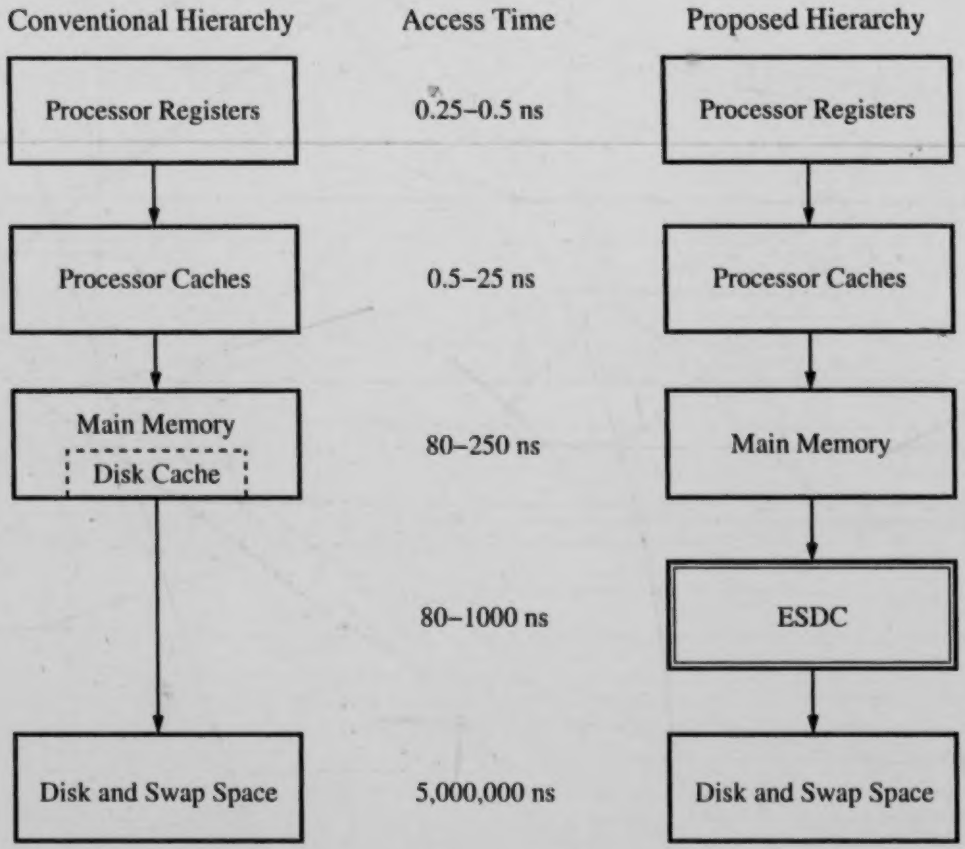

Conventional Hierarchy
Access Time
Proposed Hierarchy
Processor Registers
0.25–0.5 ns
Processor Registers
Processor Caches
0.5–25 ns
Processor Caches
Main Memory
Disk Cache
80–250 ns
Main Memory
80–1000 ns
ESDC
Disk and Swap Space
5,000,000 ns
Disk and Swap Space

Figure 4.1: The ESDC memory hierarchy adds a distinct stage to the conventional hierarchy. Memory hierarchy access times are courtesy of [25].

of a large quantity of economical, but slow, file memory. A fixed quantity of this memory is always available since ESDC has exclusive access to the memory in the new stage of the hierarchy. Replacing an operating system disk cache with slower memory is expected to impact peak performance, but the objective is to determine if ESDC will improve the overall performance of the system by filling the access time gap.

An evaluation platform is required to determine the impact of extended storage on the performance of a computer system. Several different methods that model extended storage are possible, such as by extending a cache hierarchy simulator or customizing a computer system simulator [84]. A cache model would have the advantage that a simple approach is easy to comprehend and use as a basis of a new architecture. However, extensions of simple methods often involve additional complexity, which can adversely impact the accuracy of the model [79]. Another reason to avoid basing ESDC on a cache model is the fact that extended storage is much larger and more complex than processor caches. That is, memory hierarchy simulators frequently ignore the effect of disk I/O, operating system behavior, and processor optimizations on performance results.[1] Therefore, a memory hierarchy model would not be a feasible basis for evaluating ESDC.

Evaluating extended storage involves modifying the memory hierarchy of an operating system to create an experimental platform. Fortunately, this approach minimized implementation effort. The basis for ESDC design is the fact that it is not difficult to create a model of file memory using a portion of conventional DRAM. By introducing access time penalties, this region of conventional DRAM can accurately model the performance of file memory. It must be emphasized that this design is not simply a method of emulating file memory. The evaluation platform was designed so that a bank of physical file memory could be added to a desktop computer system with minimal effort. If some page frames are marked as faulty, the bank of file memory would function as an authentic ESDC.

An extended storage disk cache was designed and implemented by modifying the Linux operating system. Linux was selected instead of other operating systems such as OpenBSD for several reasons. First, it is a public-domain operating system freely available on a number of architectures. Second, it has matured to become an enterprise-class operating system with a sophisticated memory management subsystem. Third, a large global community provides numerous resources to kernel architects, which helps to alleviate the scarcity of kernel documentation. Thus, ESDC implementation is based on the Linux 2.4.18 operating system kernel [70]. Since the implementation exploits some features specific to the Intel Pentium architecture, support for other architectures is not available at this time. A discussion of ESDC must mention some subtle operating system features to prove that ESDC achieves all design objectives. The next section will outline various components of the design and implementation of ESDC that are discussed in this chapter.

---

[1] For more information on the simulation methodology, see Chapter 5.

4.3 ESDC Design Principles

The design of ESDC requires a number of innovations to provide maximum flexibility while minimizing the impact of the implementation on the operating system. In other words, the ESDC design appears to be complex, but the implementation is remarkably simple. The approach of minimizing implementation impact promotes portability to future versions of the Linux operating system. The design of ESDC also attempts to minimize the modifications made to the Linux kernel to maintain the stability and reliability of the operating system. The goal of flexibility was addressed by creating a design that is both configurable and suitable for verification. A summary of specific ESDC design objectives follows:

High memory management ESDC is supported on systems with arbitrary physical memory sizes including systems with high memory.

Memory hierarchy integration All pages in the Linux page cache and the swap cache are considered to be ESDC pages.

ESDC page containment All ESDC pages are collected together in a contiguous region of physical memory. This requires modifications to the memory management mechanism.

Configurable performance ESDC supports emulation of physical file memory chips that have reduced performance compared with normal memory. This is accomplished with configurable performance penalties.

Caching properties of ESDC The design provides control over whether or not ESDC pages are cached by the processor caches. This requires a contiguous region of physical memory for ESDC pages.

Demand paging with ESDC Demand paging is supported: In the event of a page fault, the kernel attempts to retrieve the page from ESDC before the disk.

Metrics acquisition The design includes a facility for acquiring ESDC access metrics that are used to determine statistics such as hit and miss rates.

Implementation robustness ESDC metrics must be consistent. Several contradictions revealed some implementation flaws of the Linux kernel itself.

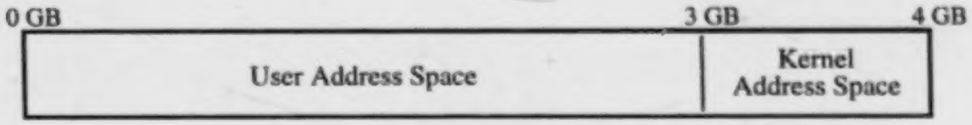


Figure 4.2: Virtual address space in Linux.

This chapter will discuss how each of the above design objectives are critical components of the overall ESDC architecture. Some topics will provide a brief background on relevant aspects of operating system memory management. Summaries of the implementation of the design objectives are also presented.

4.4 High Memory Management

An extended storage disk cache design should support systems with arbitrary file memory sizes. This objective requires a basic understanding of operating system memory management. In particular, a mechanism is needed to access the memory that can not be directly addressed by the operating system kernel. An overview of address spaces in the context of the Linux operating system will be followed by a discussion of exploiting memory zoning for ESDC. As well, this design objective involves ensuring support for arbitrary file memory sizes, which has been accomplished in this thesis by the innovative adaptation of high memory emulation.

4.4.1 Memory Address Space

Modern microprocessors are frequently equipped with multiple execution states. Linux kernels use two execution states: *user mode* and *kernel mode*. A program run in user mode can not directly access kernel data structures. The processor switches to kernel mode when the program requires that the kernel perform an operating system service.

In Linux, memory is addressed with virtual addresses in a linear address space. For 32-bit Intel architectures, the virtual address space is limited to 4 GB. The first three gigabytes of this linear address space are available for user processes while the fourth gigabyte is always mapped by the kernel (see Figure 4.2). That is, a process

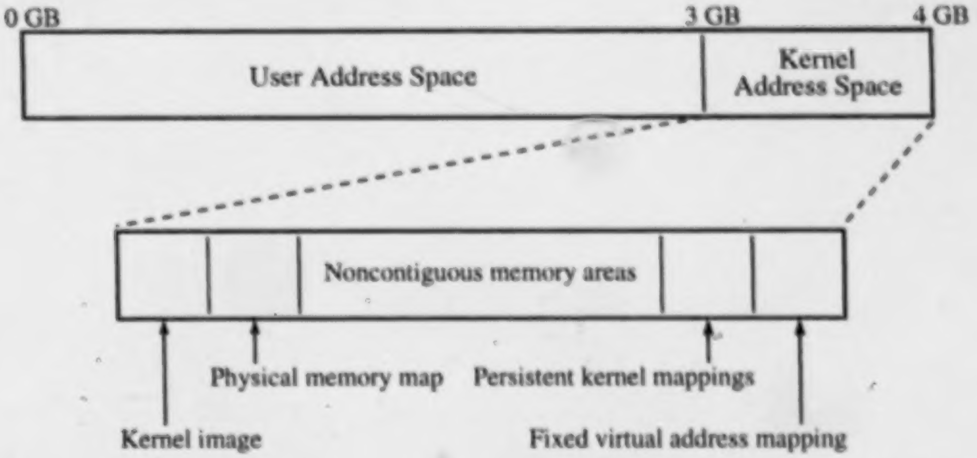

0 GB
3 GB
4 GB
User Address Space
Kernel Address Space
Noncontiguous memory areas
Physical memory map
Persistent kernel mappings
Kernel image
Fixed virtual address mapping

Figure 4.3: Kernel address space [24].

running in user mode accesses its private portion of the user address space.[2] The process only is able to access the kernel address space when it is running in kernel mode [57, p. 19].

4.4.2 Kernel Address Space

Since ESDC design involves memory management modifications, it is important to emphasize the difference between kernel and user address spaces. A user process will see a flat linear address space, but the kernel's address space is more complicated.

The kernel has access to the entire user address space as well as the one gigabyte of kernel address space. Various regions of the kernel address space are reserved for high memory management (see Figure 4.3). The kernel has room in its address space for non-contiguous virtual memory allocation. Permanent and temporary kernel mappings permit the mapping of high memory pages into low memory. Temporary kernel mappings never block the current process like permanent kernel mappings, so they are safe to use for purposes such as interrupt service routines. However, only a few temporary kernel mappings can be used at once due to the limited number of indices in the fixed virtual address map. Also shown in Figure

---

[2] A process may also access memory areas that are shared with other processes.

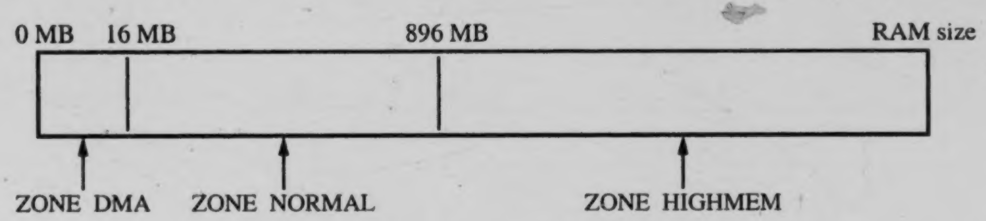


Figure 4.4: Zoning of physical memory.

4.3 is the physical memory map that enables kernel virtual addresses to directly map to physical memory addresses [8, p. 67, 256]. This mapping serves a region of physical memory known as the low memory zone. It is important to understand the Linux zoned memory architecture since it is exploited by ESDC.

4.4.3 Memory Zones

Some computer architectures have memory addressing constraints that preclude a consistent physical memory address space. Systems based on 32-bit Intel architectures can be equipped with enough physical memory so that portions of this memory are not addressable by the operating system kernel. Linux handles such memory addressing constraints by partitioning physical memory into several zones. An important zone boundary address can be found by Equation 4.1:

$$zone_{max} = addr_{kernel} - addr_{reserve} \tag{4.1}$$

where $zone_{max}$ is the highest page frame address directly addressable by the kernel; $addr_{kernel}$ is the kernel address space size (typically defined as 1 GB for Intel architectures); and $addr_{reserve}$ is the 128 MB address space reserved for high memory and non-contiguous memory allocation [8, p. 67, 256].

With these default values, $zone_{max}$ is fixed at 896 MB for 32-bit Intel architectures. That is, page frames above 896 MB can not be directly accessed by the kernel. Therefore, it is necessary to divide the physical memory address space into a zone for low memory and another zone for high memory. A small zone located at the lowest physical memory addresses is needed for compatibility with particular types of hardware. These three zones, illustrated in Figure 4.4, are named as follows:

`ZONE_DMA` This low memory zone is used by ISA-based devices that can only address the first 16 MB of RAM with Direct Memory Access (DMA).

`ZONE_NORMAL` Pages of memory between 16 MB and $zone_{max}$ are those that can be directly accessed by the kernel. These low memory pages can be linearly mapped in the fourth gigabyte of the virtual address space.

`ZONE_HIGHMEM` This zone contains pages at or above $zone_{max}$ that can not be directly accessed by the kernel using the physical memory map in the fourth gigabyte of the virtual address space.

The zoned memory architecture has a number of implications for ESDC design. First, a primary design objective is that ESDC is supported on systems with a large amount of physical memory. Consequently, high memory support is essential. Second, physical memory is partitioned into the first two zones if less than $zone_{max}$ of physical memory is available. Therefore, ESDC should still function even if no physical high memory is available. Finally, ESDC design is simplified because the high memory zone is used by default for the page cache and user processes. These implications will be discussed in more detail in later sections of this chapter.

4.4.4 High Memory Emulation

The ESDC architecture requires that high memory support be enabled regardless of the amount of available physical memory. Even if plenty of memory is available, the existing *mem* kernel boot parameter can be used to limit the amount of memory visible to the kernel. Therefore, a general mechanism was designed so that high memory emulation is enabled if less than $zone_{max}$ of memory is available. This mechanism provides the ability for adjusting the size of ESDC before the kernel boots. This is accomplished by a new kernel boot parameter, *esdc*, that permits specification of the size of the high memory zone in units of MB.

In Linux 2.4, the high memory zone is enabled through the use of several kernel configuration parameters, as shown in Table 4.1. For systems with more than

Table 4.1: High Memory Configuration Parameters

Parameter	Description
`CONFIG_HIGHMEM`	Enable support for high memory
`CONFIG_HIGHMEM4G`	Enable memory accesses between 1 and 4 GB of RAM on a 32-bit CPU
`CONFIG_DEBUG_KERNEL`	Enable advanced kernel debugging features
`CONFIG_DEBUG_HIGHMEM`	Enable high memory emulation

$zone_{max}$ of memory, only the first two parameters are needed to activate an actual high memory zone. However, an ESDC architecture should not be limited to systems with substantially more memory than $zone_{max}$. Therefore, high memory emulation is needed for systems with less that $zone_{max}$ of memory. Emulation of high memory is activated by setting all of the configuration parameters in Table 4.1. Furthermore, it is necessary to specify the size of an emulated high memory zone, although this was not indicated in any available Linux 2.4 kernel documentation. Further details can be found in Appendix A.

4.5 Memory Hierarchy Integration

The design of an extended storage disk cache requires that a new stage be introduced into the memory hierarchy. Instead of designing a redundant hierarchy stage, ESDC is an adaptation of the Linux virtual disk cache. This creates a more distinct stage in the memory hierarchy since file memory is not available for any uses other than disk caching. As shown in Figure 4.1, Linux uses a portion of conventional DRAM memory as a disk cache of pages. That is, the design is based on conversion of the Linux page cache into ESDC.[3] While the functionality of ESDC is similar to the page cache, some virtual disk caches are excluded from extended storage.

In Linux 2.4, the *page cache* is a disk cache containing pages that correspond to several logically contiguous blocks of files. Page replacement is handled by an algorithm that approximates the least-recently-used (LRU) page replacement policy.

---

[3] As explained later in this chapter, the swap cache is a subset of the page cache.

Unfortunately, the kernel does not keep an LRU list of the data pages belonging to a process that are in memory. Therefore, it is difficult to collect the data necessary for a cost/benefit analysis [10, 35].

The page cache was converted to ESDC instead of adding a new hierarchy stage. This approach was chosen for several reasons. First, an independent disk cache implementation would involve the duplication of the functionality of the Linux page cache. Smith has explained how multiple caches serving the same function should be avoided since they add additional overhead and introduce possible side effects that reduce performance [63]. Second, a redundant hierarchy stage would impact a host of optimizations related to I/O latency reduction [79]. For example, a disk read-ahead should not be performed when reading from a redundant disk cache stage below the page cache stage in the hierarchy. The read-ahead is unnecessary because there is no performance penalty for retrieving pages from a disk cache at different times as there is with disk media [10]. Third, the existing page cache implementation already has been optimized to work efficiently for a variety of operating system workloads. Finally, page cache adaptation does not violate the principle of minimizing the modifications made to the Linux kernel. Even minor modifications have the potential to reduce the performance or stability of the operating system. The overhead introduced by management of additional metadata can significantly degrade operating system performance. Castro and others have noted how simple kernel enhancements offered better results than more complex algorithms [10].

In spite of the above advantages, converting the page cache to ESDC has a negative impact on peak I/O performance. Instead, an alternative architecture preserves the existing virtual disk cache and introduces extended storage as a new hierarchy stage below main memory. This approach has the benefit of isolating the inferior access time of file memory. However, it would introduce additional complexity and overhead into the kernel, so a mechanism of bypassing the extended storage stage for blocking I/O operations would be useful. Such a design requires radical changes to the operating system and is not investigated in this thesis.

4.5.1 Hierarchy Properties

To understand how ESDC functions as a memory hierarchy stage, it is necessary to discuss four memory hierarchy attributes: page placement, page identification, page replacement, and write strategy [25].

Since ESDC is based on the page cache, it is important to know how a page is added to the page cache to determine where it should be placed within the cache. Within kernel space, a page does not have to be duplicated to be added to the page cache. Instead, a reference counter in the page descriptor associated with the page is simply incremented when a page is added to the page cache. If this counter ever becomes unity, then the page can be removed from the page cache. When a page has been added to the page cache, a field of the corresponding page descriptor points to the kernel object that owns the page. This `mapping` field is not defined if the page is not in the page cache. This means that if a page in memory is mapped to an inode of a file on disk, then it is in the page cache [8, p. 382, 480]. Since ESDC is based on the page cache, pages with a mapping field are ESDC pages. Therefore, because a page anywhere in high memory can become part of the page cache, ESDC is *fully-associative*.

Various components of the Linux memory management subsystem need to determine if a page is present in ESDC as fast as possible. A hardware cache usually searches all possible address tags in parallel to determine if a line is present in the cache. The Linux kernel does not have the luxury of a parallel search. Instead, a hash table of page descriptor pointers is maintained to improve the performance of page cache searches. For example, when a file is read, the page cache is searched to determine if the pages associated with the inode of that file are already in the page cache. Similar searches are performed for other functions involving performing read-aheads and generic file input and output. Since pages in the page cache also can be in the swap cache, the same search algorithm also determines if a page had been paged to disk and was added to the swap cache.

ESDC employs the *least-recently-used* (LRU) page replacement policy. When a page is to be added to the page cache and no free page frames are available, an existing page frame in the page cache must be replaced with the new page. The LRU page replacement algorithm is based on temporal locality; pages recently accessed will likely be accessed soon, so the least-recently-used page should be the candidate for replacement. The kernel implements the LRU algorithm using two lists of pages. The *active list* records the pages most recently accessed while the *inactive list* is a collection of pages that have not been accessed for a while. The kernel uses several page access flags and periodically moves pages between the lists in response to the activity level of the entire system [8, p. 563]. This LRU implementation is suitable for ESDC because it has been optimized to work efficiently with other components of the Linux operating system.

When a page is written to ESDC, the page will eventually be replicated on a disk or device. The page cache functions as a cache with *delayed writes* (see Section 2.6). All writes to disk first are written to the page cache. Unlike in a conventional cache, backing pages to disk only when the page cache reaches capacity would compromise reliability. Writing pages to the disk immediately would offer high reliability at the expense of performance. Therefore, a kernel synchronization thread is responsible for periodically flushing dirty block device data to disk. It acts as a write-back daemon to improve performance while introducing some reliability exposure. In Linux, this thread is run every 30 seconds to flush data to block devices.[4] Similar mechanisms exist in other implementations of Unix [1, 13].

4.5.2 Pages Excluded from ESDC

In Linux 2.4, not all disk I/O makes use of the page cache. The *buffer cache* is an auxiliary disk cache that is used as a cache for single disk blocks, or *buffers*. Most buffers in an Ext2 file system are 1 KB in size because each block contains data belonging to two disk sectors. As shown in Table 4.2, most file system I/O passes through the page cache [8, p. 475]. However, accesses to file system metadata such as superblocks use the buffer cache instead of the page cache. Nevertheless, buffers

---

[4] A comparison of file system workloads in [55] indicated that the average block lifetime is significantly longer than 30 seconds for some workloads. Section 2.6 discusses the impact of longer write delays on performance and reliability.

Table 4.2: Page Cache and Buffer Cache

Kernel Function	Cache	Operations
bread()	Buffer	Read an Ext2 superblock or inode
generic_file_read()	Page	Read an Ext2 directory or regular file
generic_file_write()	Page	Write an Ext2 directory
generic_file_read()	Page	Read a block device file
generic_file_write()	Page	Write a block device file
filemap_nopage()	Page	Access a memory-mapped file
brw_page()	Page	Access to a swapped-out page

still can be used to store block data in *buffer pages* that belong to the page cache. In Linux 2.4, the buffer cache and page cache are intertwined when buffer pages are involved [8, p. 487]. I/O operations on a page without buffers mark the page as clean or dirty, but the buffers themselves are marked as clean or dirty instead of the associated buffer pages. Buffer pages containing dirty buffers must first be migrated to backing store before they can be freed [10].

In this work, the Linux page cache is converted to ESDC while the buffer cache implementation remains unmodified. This simplification was chosen to ensure that the experimental results are accurate and that kernel modifications are not double-counting disk cache access metrics. Furthermore, this simplification is necessary since forcing metadata I/O through ESDC could affect operating system performance more drastically than the case where ESDC replaces only the page cache. Metadata overhead can severely affect the overall performance of a system under memory pressure [10].

Pages for five different types of I/O operations may exist in ESDC. Four of these types correspond to the different I/O operations shown in Table 4.2. The page cache may also include pages belonging to a shared memory region for interprocess communication (IPC). Therefore, ESDC will cache pages associated with block device files, directories, memory-mapped files, and swapped-out pages. Pages that must not appear in the page cache include those that are managed by *self-caching appli-*

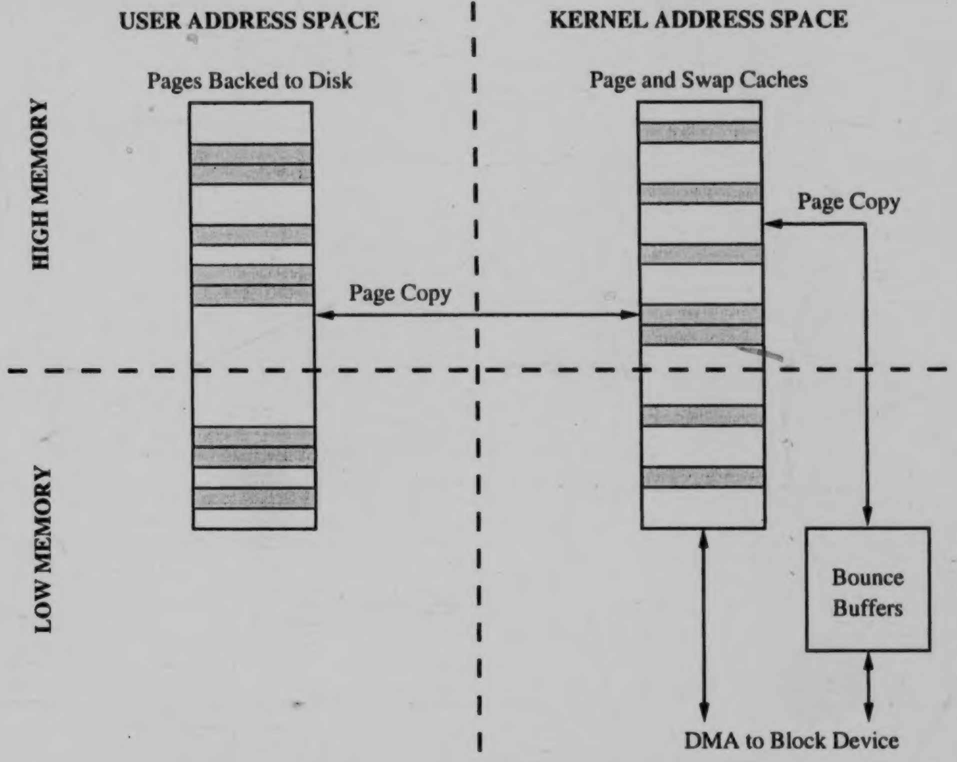

USER ADDRESS SPACE
KERNEL ADDRESS SPACE
HIGH MEMORY
LOW MEMORY
Pages Backed to Disk
Page and Swap Caches
Page Copy
Page Copy
Bounce Buffers
DMA to Block Device

Figure 4.5: Original page cache architecture with page dispersion.

cations. For example, some database servers provide their own disk caches that are more suitable to database queries than a general page cache. To accommodate such requirements, the kernel offers a datapath for direct I/O transfers that bypasses the page cache [8]. However, self-caching applications should be special cases to discourage the use of application-based disk caching systems, because some attempts at disk caching may have questionable effectiveness [63].

4.6 ESDC Page Containment

Since the physical memory location of pages is not changed when the pages are added to the page cache, most pages in the page cache are not in a contiguous region of physical memory. For the ESDC architecture, it is necessary to restrict page cache pages to a contiguous memory region. A distinct ESDC region pro-

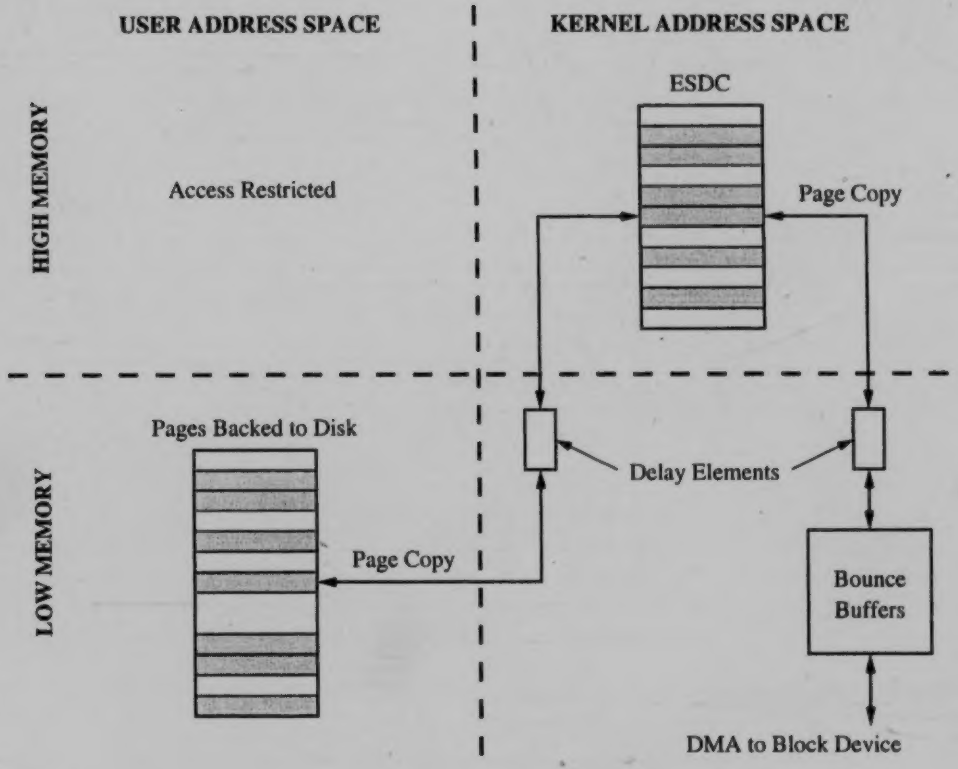

USER ADDRESS SPACE
KERNEL ADDRESS SPACE
HIGH MEMORY
LOW MEMORY
Access Restricted
ESDC
Page Copy
Pages Backed to Disk
Delay Elements
Page Copy
Bounce Buffers
DMA to Block Device

Figure 4.6: ESDC architecture featuring page containment.

vides several advantages such as support for large memory capacities. The design of the ESDC page containment mechanism is based on the existing zoned memory architecture. After a thorough analysis of memory allocation in Linux, page containment requires remarkably few kernel modifications. Finally, page containment in high memory handles performance issues in spite of the overhead of copying pages between zones.

4.6.1 Architectural Considerations

When Linux 2.4 is configured to support high memory, the page cache and pages belonging to user processes reside primarily in the high memory zone by default. As shown in Figure 4.5, some of these pages also can appear in low memory zones. Pages belonging to the page and swap caches are allocated primarily in high memory, but also can be allocated in low memory zones. That is, the low memory zones act as fallback zones for memory allocation whenever the high memory zone is near capacity. Other operating system data structures normally reside in the low memory zone. Therefore, in the unmodified kernel, only user pages and pages in the page cache are allocated in high memory. However, these pages can be interleaved with each other and can appear in any memory zone.

For this work, the kernel is changed so that page cache is contiguous and contained entirely within high memory, as illustrated in Figure 4.6. For purposes of evaluating ESDC, the presence of a contiguous region of ESDC memory in the high memory zone offers a number of advantages. First, by locating ESDC memory exclusively in the high memory zone, it is simple to introduce performance penalties, or *delay elements*, to the transfer of pages between normal memory and potentially slower ESDC memory (see Section 4.7). Second, the contiguous region of memory allows adjustments to be made to the caching properties of ESDC (see Section 4.8). Third, since the contiguous region of ESDC memory is located in the high memory zone, ESDC capacities of tens of gigabytes are possible in spite of the 32-bit architecture's addressing limitations. Finally, if physical file memory becomes available, it is a reasonable assumption that ESDC would reside at physical memory addresses higher than those used for conventional DRAM. Therefore, it is necessary to relocate all user pages to the normal memory zone so that only ESDC pages remain in the high memory zone.

ESDC pages are not directly accessible by processes since ESDC is based on the page and swap caches within the kernel address space. Likewise, the kernel is not able to directly access a page in the user address space because it is not straightforward to determine if the page is resident. Therefore, both page cache pages and ESDC pages need to be copied between user address space and kernel address space (see Figures 4.5 and 4.6). Whenever a user page needs to be written to disk, the page must be copied from user space to kernel space. Similarly, a page with a mapping to a file on disk must be copied from kernel space to user space before the user process can access the data. It is important to emphasize a major benefit of this architecture: User processes are not directly affected by the reduced performance of ESDC memory since ESDC page frames are never mapped to the user address space. In addition, all page replications were present in the original kernel to support locating a page cache in high memory. In this manner, ESDC design exploits existing kernel features rather than copying pages to new data structures.

The transfer of pages between processes and ESDC memory that has just been discussed does not include the interface between ESDC and disk. This second interface involves copying buffers associated with pages between ESDC and bounce buffers.

4.6.2 Bounce Buffers

ESDC design relies, in part, on some obscure and inefficient high memory management techniques. Since page frames in high memory cannot be directly accessed by the kernel, they have to be mapped into the kernel address space. This is accomplished by using non-contiguous memory allocation, permanent kernel mappings or temporary kernel mappings. However, not all I/O devices are able to address high memory. Linux 2.4 uses *bounce buffers* to solve this problem. In spite of the fact that using bounce buffers is an inefficient solution, they help to simplify ESDC implementation. Moreover, other extended storage hierarchy designs enforce similar restrictions on extended storage I/O operations. In [53], for example, it is not possible to directly transfer data between extended storage and a block device.

Bounce buffers appear in the high memory datapath as shown in Figure 4.6. A legacy device that can not access high memory performs I/O operations on a bounce buffer that is allocated in low memory. A bounce buffer functions as a bridge between low memory and high memory. When writing to a device, data is copied from high memory when the bounce buffer is created. When reading from a device, a callback function copies the data from the bounce buffer to high memory. Even though buffer page copying introduces significant overhead, this approach provides access to high memory that would otherwise not be available

[24]. Bounce buffers offer a major advantage for ESDC since they conveniently assist in the provision of configurable performance penalties for evaluating ESDC (see Section 4.7).

4.6.3 Page Allocation

After spending many days making a determined effort to comprehend Linux memory management architecture, page containment was achieved by remarkably simple kernel modifications. Several different kernel modifications would give ESDC pages exclusive access to high memory. Various changes to the design of the memory management subsystem are possible, such as changing the structure of the lists that manage the memory zones. However, the design alternative that minimized the number of kernel changes involved modifying the page allocation flags. These flags are known as the *get free page* (GFP) flags.

The various memory zones in Linux 2.4 serve different purposes. When page frames are allocated, various GFP bitmasks control the desired memory zone. For example, the `__GFP_HIGHMEM` determines whether or not a page will reside in the high memory zone. To support ESDC page containment in high memory, the use of this flag needs to be restricted. In particular, this flag is used by several memory allocation function calls responsible for allocating memory from the virtual address space. This memory is mapped to kernel space as a contiguous range of virtual addresses and is not visible from user space. Therefore, such instances of the `__GFP_HIGHMEM` flag must be removed to help ensure that the high memory zone is only used for page cache allocations. A detailed analysis of memory allocation flag usage in the kernel and its impact on ESDC design is in Section A.1.2.

ESDC design also depends on the fact that pages associated with inodes are allocated in the high memory zone. An inode object in the kernel contains data structures that manage block device files. In particular, one of the purposes of these data structures is to identify whether or not a page is in the page cache [8, p. 477]. During initialization of these data structures, a bitmask is used to set the memory allocation flags for the owner of the pages associated with the inode (see Section

A.1.2). If this bitmask contains the `__GFP_HIGHMEM` flag, then the pages will be allocated in the high memory zone. Consequently, such pages that are contained in high memory are known as ESDC pages.

4.7 Configurable Performance

ESDC is intended to model file memory with reduced performance relative to conventional DRAM. For slow file memory to offer a benefit, it must not impact the performance of the process address space. The crux of the idea is that pages must be copied to or from ESDC, but processes or device drivers do not have direct access to ESDC pages. This objective was accomplished by basing ESDC design on the page and swap caches and locating ESDC in high memory. Therefore, a method of applying access time penalties is needed to emulate a physically slower memory, since, for this work, file memory is simply a region of conventional DRAM. One solution would be to calibrate a delay function to increase the access time to an ESDC page. This approach involves non-deterministic kernel behavior so the accuracy of the model is questionable. Instead, modifying the kernel to perform repeated accesses to an ESDC page would be a more reliable solution.

4.7.1 Design Alternatives

There are several methods for introducing performance penalties to artificially reduce the performance of file memory relative to conventional DRAM. One method of calibrating a timer function to measure the page copy duration is problematic because the elapsed time of page copies is not deterministic. When copying from user space, for example, a page fault could occur if the page was not resident. Such page faults, or interruptions by the process scheduler during the delay loop, would result in unpredictable delay durations. Another possible solution would be to delay the copies of the individual bytes or words of the page. However, this is not feasible because the functions that copy pages to or from user space are highly optimized assembly functions and it would be more difficult to introduce fine-grained performance penalties.

The best solution for introducing penalties involves repeated calls to the functions that copy pages between ESDC and conventional DRAM. The repeated replication of pages between user space and ESDC is represented by the left delay element in Figure 4.6. The right delay element is introduced between ESDC and the bounce buffers. This method allows for fine-grained penalties, but requires that ESDC memory not be cached by the processor caches (see Section 4.8). The impact of uncached ESDC memory on performance is evaluated in Section 5.4.2.1.

4.7.2 Implementation Issues

To implement ESDC performance penalties, it was necessary to determine which functions transfer pages between user space and the page cache. This was not a straightforward exercise because the kernel features a large number of functions involved with file I/O and the page cache. Eventually, it was found that the functions involved with generic block writes copy pages from user space to kernel space using `__copy_from_user()`. To copy pages in the other direction for file reads, a different function calls `__copy_to_user()`. Detailed callgraphs are available in Section A.1.3.

Since pages are transferred between ESDC and block devices via the bounce buffers, delay elements must be added to a second set of functions. One candidate function that copies buffers located at a high memory address to a bounce buffer is named `copy_from_high_bh()` and is used when data is written to a block device. The other function, `copy_to_high_bh_irq()`, copies buffers in the other direction and is used when data is read from the device. Fortunately, bounce buffers transfer buffer pages, so both pairs of copy functions operate on 4096-byte blocks of data.

ESDC performance penalties were implemented at these four critical points in the kernel memory management system. When a file I/O operation proceeds for a page, one of the above functions is called multiple times according to an access time penalty ratio specified via the `proc` file system (see Section 4.10). The normalized access time ratio of 1 implies a single page copy and represents file memory with the same performance as conventional DRAM. However, a ratio of 2 would result in two copies of the affected page to the same destination address. This access time ratio emulates file memory with twice the access time of conventional DRAM. Estimates of partial page copies permit fine-grained ratios, such as 2.5. Since ratios less than 1 are not permitted, the page is always copied at least once.

This method has a major, but surmountable, drawback. Multiple copies of a single page from the same source to the same destination will likely involve the processor caches. Repeated copies of cached data will drastically skew the effect of the penalty factor. That is, if the page is cached after the first copy, subsequent page copies would require much less time. For example, an access time ratio of 6 may only double ESDC access time rather than increase it by a factor of six. To ensure accurate performance penalties during ESDC experiments, the operating system is directed to prevent the processor caches from caching ESDC memory. Controlling the caching properties of areas of memory is discussed in detail in Section 4.8.

4.8 Caching Properties of ESDC

Since the ESDC memory area is contiguous, it is possible to control how this memory is cached by the hardware caches. This is necessary for two reasons. First, one method of adjusting ESDC performance penalties requires that the hardware caches do not cache lines from ESDC memory (see Section 4.10). Second, large ESDC memory sizes may require memory hardware to be accessible via an I/O bus or backplane bus rather than the memory bus. The performance of such memory is limited by the low bandwidth of the bus interface. An example of an external memory includes a solid-state disk with gigabytes of DRAM. Memory accessed via the PCI or USB buses normally is not cached by the CPU caches. ESDC memory is able to model slow external memory if it is configured to be uncachable by the processor caches.

The Intel architecture provides a mechanism for specifying the type of caching for various regions of memory [32]. The caching methods are known as *memory types* and include the following:

Uncachable Memory locations are not cached and all memory accesses are executed in order.

Write Combining Memory locations are not cached, but writes may be delayed and speculative reads are permitted.

Write-through Memory reads and writes are cached, but writes to a cache line are also written to memory.

Write-back Memory reads and writes are cached, but writes to a cache line are not immediately updated in memory.

Write Protected Memory reads from the cache proceed as usual, but writes invalidate the associated cache line.

The memory types of various address ranges in system memory can be specified by a set of registers known as the *Memory Type Range Registers (MTRRs).* They are normally used to optimize operations for unique regions of memory, such as frame-buffer memory and memory-mapped I/O devices. However, for this work, MTRRs are used to adjust the caching method of ESDC memory.

MTRR registers impose several restrictions on the number and sizes of the address ranges that they specify. For physical memory addresses above 1 MB, only eight MTRR ranges can be specified using sixteen MTRR registers.[5] The kernel defines two MTRR ranges by default. One range sets all available physical memory to the default memory type of write-back. The other MTRR range reserves an 8-MB block of memory starting at physical address 2048 MB, a block that ESDC must avoid. Therefore, only six MTRR ranges are available for use by ESDC. Another limitation involves the fact that the sizes of the variable MTRR ranges must be of length 2^n. As well, the base addresses of these ranges must be aligned on a 2^n boundary, where the boundary is an integer multiple of the size of the range.

---

[5] If necessary, a *page attribute table* could be employed to provide functionality equivalent to an unlimited number of MTRRs.

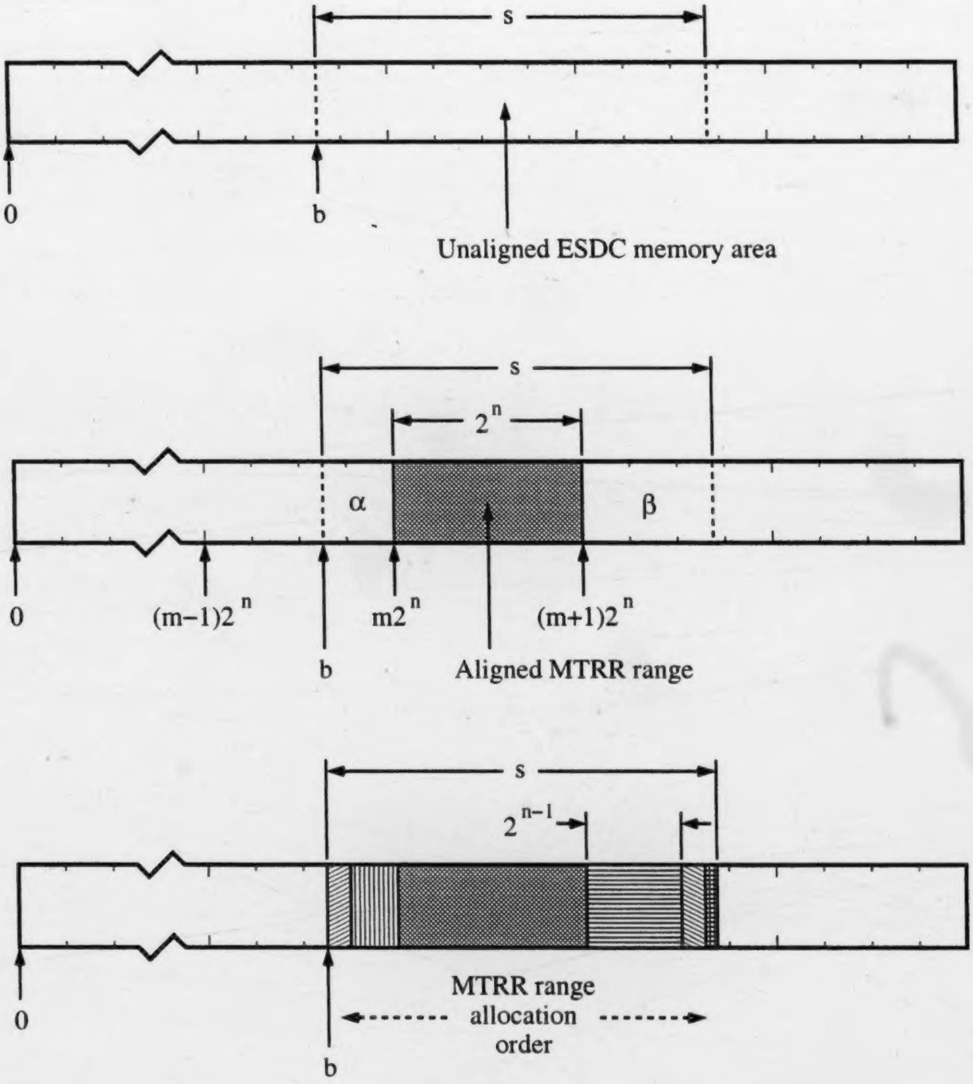

s
0
b
Unaligned ESDC memory area
s
2^n
α
β
0
(m−1)2^n
m2^n
(m+1)2^n
b
Aligned MTRR range
s
2^{n−1}
0
b
MTRR range allocation order

Figure 4.7: Algorithm to create aligned MTRR ranges within ESDC.

To satisfy the above constraints, a new algorithm was designed to partition ESDC memory into aligned MTRR ranges.[6] This algorithm was inspired by the method of memory allocation used by the Buddy System (see Section 3.4.2) [41, 42]. The algorithm determines the largest aligned MTRR range that fits within the potentially unaligned ESDC memory area, as shown in Figure 4.7. Using recursion, the remaining portions α and β are subdivided into one or more smaller aligned MTRR ranges using the same algorithm. During each step of recursion, the boundaries of a previous aligned MTRR range become a boundary of a subsequent conterminous MTRR range. Hence, the remaining MTRR ranges are defined first for α and then for β in the order shown in Figure 4.7.

---

[6] It should be noted that ESDC remains a contiguous area of memory. The partitioning described here is only needed for the MTRR cache control mechanism.

For an unaligned memory area of size s starting at base address b, the aligned MTRR range must satisfy three constraints:

$$2^n \leq s \quad | \quad n \in max\{p \leq i < \infty,\ i \in I\} \tag{4.2}$$

$$b \leq m2^n \quad | \quad n \in max\{p \leq i < \infty,\ i \in I\},\ m \in I \tag{4.3}$$

$$(m+1)2^n \leq b+s \quad | \quad n \in max\{p \leq i < \infty,\ i \in I\},\ m \in I\,. \tag{4.4}$$

By Equation (4.2), the MTRR range size 2^n is the largest value between the page size 2^p and the ESDC memory size s. Equation (4.3) restricts the base address of the aligned MTRR range to an aligned boundary address. This boundary address must satisfy (4.4) to ensure that the aligned MTRR range remains within the ESDC memory area.

To reduce the number of kernel modifications, the MTRR alignment algorithm was not implemented in the kernel. Instead, it was implemented as a recursive function in `mando.pl`, a Perl benchmark automation utility (see Section C.2). The implementation utilized the `proc` file system interface for MTRRs. That is, the function adds one or more additional records to `/proc/mtrr` to create MTRR ranges for ESDC memory. ESDC benchmarks make use of this implementation when evaluating uncached ESDC memory.

4.9 Demand Paging with ESDC

ESDC design requires a detailed analysis of the page fault exception handler so that ESDC memory is used correctly during demand paging. *Demand paging* is a method of memory allocation where page frames are allocated at the last possible moment to improve performance. Rather than swapping entire processes into

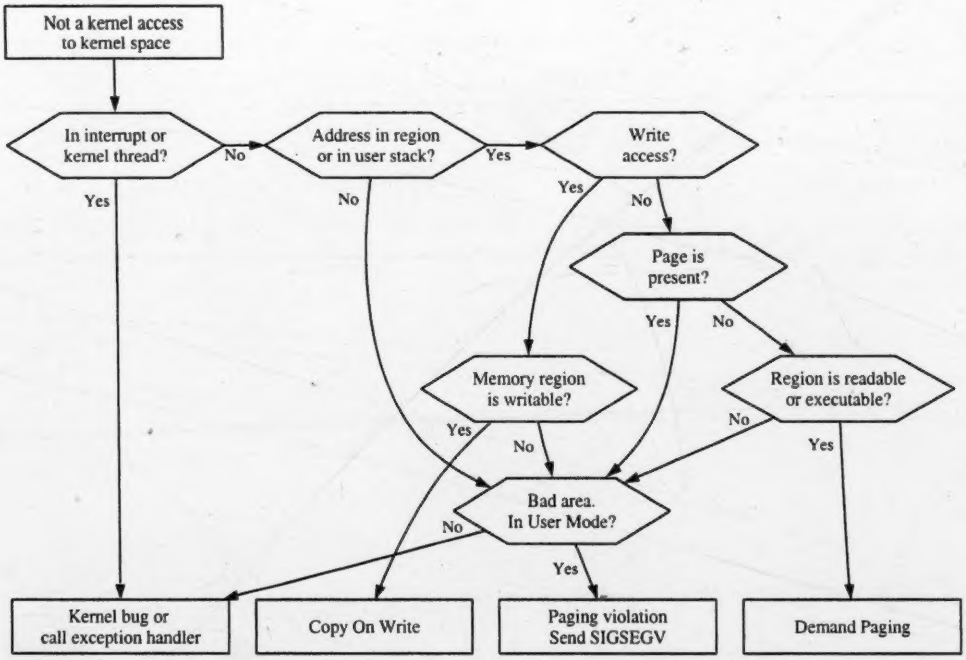

Not a kernel access to kernel space
In interrupt or kernel thread?
No
Address in region or in user stack?
Yes
Write access?
Yes
No
Yes
No
Page is present?
Yes
No
Memory region is writable?
Region is readable or executable?
No
Yes
Yes
No
Bad area. In User Mode?
No
Yes
Kernel bug or call exception handler
Copy On Write
Paging violation Send SIGSEGV
Demand Paging

Figure 4.8: Simplified overview of page fault exception handling in Linux [8].

memory from backing store, individual pages are paged in when needed.[7] When a process attempts to access a page and triggers a page fault, the page may have to be paged in from backing store [22]. This operation, a *major fault*, is an expensive last resort. It is preferable for the kernel to incur a *minor fault*, which does not block the execution of the current process.[8] Since page faults can be caused by a variety of conditions, certain components of the page fault exception handler needed to be modified to support ESDC.

The page fault exception handler must identify the cause of the exception before it can service the fault. As shown in Figure 4.8, the current state of the process, the location of the memory access in the virtual address space and the properties of the affected region of memory help to determine how the page fault will be handled. Many of the alternatives in Figure 4.8 depend on the protection flags of memory regions. *Memory regions* are non-overlapping sets of pages of memory.

---

[7] In Linux, *swapping* individual pages to or from disk is synonymous with *paging*.

[8] Page faults also can be caused by illegal memory accesses or programming errors (see Figure 4.8).

They are used to identify virtual memory areas such as the heap of a process or a memory-mapped file. The region's pages may be unallocated, present, or paged out to backing store [24, p. 47]. When a page fault was caused by a legal memory access, the page fault handler will allocate a new page frame through one of two methods: demand paging or copy on write (COW). The demand paging algorithm has a number of implications for ESDC, while COW is more straightforward.

4.9.1 Demand Paging

For demand paging, a page may not be present in memory for two reasons. First, it was never accessed by a process. Second, the page was previously accessed by the process, but the data has since been paged or swapped to disk [8, p. 293]. One of the following alternatives is handled based on the properties of the page table entry.

Missing page never accessed To handle a page fault when the page was never accessed, it must be determined whether or not the page had been mapped to a file. The two cases have different implications for ESDC:

1. **Page is memory-mapped:** *Memory-mapping* refers to the situation where a memory region is associated with a portion of a file or device. It is essential that ESDC support memory-mapping because it is a common method of accessing files. In fact, workload traces have indicated that a greater number of processes use memory-mapped files instead of performing regular I/O operations [55]. Therefore, keeping memory-mapped pages in memory as long as possible will reduce associated ESDC miss rates.

   When a file is memory-mapped, none of its pages will be in memory initially. These pages will only be brought in once a process tries to access them. A new page mapping needs to be created for these missing pages; that is, a new page needs to be allocated for each page read from disk. As discussed in Section 4.5, once a page mapping is established, any disk

I/O on the page will automatically use ESDC memory as a cache. Therefore, no kernel modifications are necessary in the `filemap_nopage()` function (or similar device driver functions) to support ESDC.

2. **Page is anonymous:** A page that was never accessed may not be mapped to a file on disk. Such a page is known as an *anonymous page*. If the page fault was caused by a read access, a special read-only page containing only zeros is mapped to the process. Otherwise, a new page initialized to zeros is allocated when the anonymous page is accessed for the first time. As explained in Sections 4.6 and A.1.2, anonymous pages are not allocated in ESDC memory, even when backed by swap. This required a minor change to the function `do_anonymous_page()`.

Missing page swapped out When a page has been paged to backing store and it was accessed by a process, the demand paging algorithm has one last chance to incur a minor fault. It first examines the swap cache to check if the desired page is located in memory. A *swap cache* is a collection of shared page frames that have been replicated to one or more swap areas.[9] The swap cache is considered to be a subset of the page cache, since pages in the page cache are considered to be in the swap cache when particular page descriptor fields are set [8, p. 546]. If a major fault is necessary, a new page needs to be allocated for the data read in from disk. This page must be located in high memory to support ESDC page containment (see Section 4.6). Since the demand paging function `read_swap_cache_async()` already allocates the new page in high memory, no changes are necessary for this case. Note that ESDC is not intended to act as a paging device that is the primary backing store for pages backed by swap. Instead, a future extended storage architecture could include such support in addition to caching file-backed pages. More information on ESDC kernel modifications related to demand paging can be found in Table A.2.

---

[9] The swap cache also helps the kernel avoid a number of race conditions, but this topic is beyond the scope of this thesis.

4.9.2 Copy On Write

In early implementations of Unix, the entire address space of a process was duplicated when it forked a child process. This operation was expensive due to high memory accesses, poor cache utilization and possible paging from disk. *Copy on write (COW)* is a more efficient approach that shares pages between the parent and child processes. The associated page table entries are read-only so that a write access will trigger a page fault. On such a page fault, new page frames are allocated for the child process by the page fault handler [24, p. 74]. The functions that allocate such pages required minor modifications to support ESDC. Since these pages are intended for user processes, they do not belong in ESDC memory. A detailed explanation of the restrictions placed on ESDC memory is discussed in the next section.

4.10 Metrics Acquisition

To assist with ESDC design verification and evaluation, it is necessary to access ESDC operating metrics from a running kernel. Several mechanisms are available for monitoring the activity of ESDC. Note that it is not feasible to embed `printk()` statements throughout the kernel source to view runtime values of memory management data structures. Even if such messages are directed to system log files, the volume of text produced becomes unmanageable. Worse, the high message rate exceeds the capacity of the system logging daemons so that blocks of messages often never appear in the system log files. Instead, it is necessary to utilize and extend the `proc` file system for acquiring ESDC metrics.

4.10.1 ESDC and the `proc` File System

The `proc` virtual file system is a mechanism for monitoring or adjusting selected kernel parameters. It is not a file system in the conventional sense; it is an interface between the kernel and certain user processes. That is, `proc` files display the runtime values of kernel parameters when they are read or allow kernel parameters to

Table 4.3: ESDC Metrics

(a)	number of I/O requests
(b)	number of copies to high memory from bounce buffers
(c)	number of copies from high memory to bounce buffers
(d)	number of pages copied from high memory to user space
(e)	number of pages copied to high memory from user space
(f)	number of dirty mapped pages written to backing store
(g)	number of anonymous pages with buffers
(h)	number of pages still in page cache with buffers freed
(i)	number of pages with buffers that can not be released
(j)	number of attempts to swap out for anonymous process pages
(k)	number of pages removed from page cache to reclaim memory
(l)	number of pages removed from page cache by truncate() system call
(m)	number of pages freed from high memory by the buddy system
(n)	number of pages allocated in high memory by the buddy system

be modified [57].

New facilities have been added to the `proc` file system to control and monitor ESDC activity. The `/proc/sys/vm/esdc_ctl` virtual file allows for ESDC performance penalties to be adjusted dynamically by a user process, as discussed in Section 4.7. A modified version of the existing `/proc/meminfo` file offers more detailed information regarding high memory management. A new virtual file, `/proc/esdc`, provides runtime access to a number of ESDC properties. The metrics in this virtual file are cumulative snapshots of custom atomic variables created for ESDC.[10] For example, when a page is copied to ESDC, an atomic counter is incremented. These atomic variables hold cumulative values for the ESDC metrics shown in Table 4.3. Most of the ESDC metrics are used for verification of correct ESDC functionality. Some are used to identify unusual kernel behavior caused by undocumented race conditions, which will be discussed in Sections 4.11 and 5.4.3. Regardless of the purpose, the effect a particular event has on the kernel can be determined by the differences between two sets of metrics that bound the event in time.

---

[10] An atomic variable is accessed or changed by atomic operations.

4.10.2 ESDC Access Statistics

A number of metrics are used in the calculation of ESDC access statistics. The two ESDC interfaces described in Section 4.7 are directly involved with access metrics. The functions that are used to incur performance penalties are excellent points to update metrics for reads and writes. The `/proc/esdc` virtual file records the number of pages written to or read from ESDC by user processes. Likewise, the number of pages transferred between ESDC and disk can be monitored at the bounce buffer page transfer interface. ESDC metrics are added to the buddy system implementation to verify that the number of pages allocated to or freed from ESDC is consistent with other metrics.[11]

ESDC hit and miss rates correspond to accesses that involve page transfers to or from ESDC. It is important to distinguish ESDC page accesses from accesses to the page cache hash table. As outlined in Section 4.5, the kernel uses this hash table to quickly determine if a page is present in the page cache.[12] ESDC hit and miss rates are not measured directly from page cache metrics; instead, they are computed from ESDC page access metrics. The number of ESDC write access hits since power up is as follows:

$$w_{hit} = w_{esdc} - w_{disk} \,. \tag{4.5}$$

The number of writes from a process to ESDC is w_{esdc} and the number of writes from ESDC to disk via the bounce buffers is w_{disk}. The number of hits for ESDC read accesses is a similar calculation:

$$r_{hit} = r_{esdc} - r_{disk} \,. \tag{4.6}$$

The total number of access to ESDC is simply the sum of the number of reads and writes to ESDC. Using this fact and the above definitions for w_{hit} and r_{hit}, the ESDC hit and miss rates are defined by Equations (4.7) and (4.8), respectively. These ESDC statistics are expressed as a percentage:

---

[11] Note that ESDC memory includes both the page cache and the swap cache. Therefore, ESDC metrics represent both page cache and swap cache activity, as discussed in Section 4.9.

[12] More information on monitoring the page cache hash table is presented in Section A.1.4.

$$H = \left(\frac{w_{hit} + r_{hit}}{w_{esdc} + r_{esdc}}\right) 100\% \tag{4.7}$$

$$M = \left(\frac{w_{disk} + r_{disk}}{w_{esdc} + r_{esdc}}\right) 100\%\,. \tag{4.8}$$

Note that the sum of the terms w_{disk} and r_{disk} represents all types of ESDC misses. These misses include compulsory misses (misses caused by initial accesses to pages not in ESDC) as well as capacity misses (retrieval of discarded ESDC pages). There are no conflict misses since ESDC is fully-associative. Calculated with Equations (4.7) and (4.8), ESDC hit and miss rates are available by accessing `/proc/esdc`.

4.11 Implementation Robustness

The implementation of ESDC must provide accurate metrics to increase one's confidence in the validity of the experiments. One method of ensuring accuracy was to develop correlations between ESDC metrics and kernel memory management metrics. These two sets of metrics measure the same kernel properties using different mechanisms. Any inconsistency will indicate potential problems with either the ESDC implementation or the kernel itself.

ESDC is implemented within an operating system that is still under development. ESDC implementation was more challenging than initially expected due to the complexity of the Linux 2.4 kernel. Other researchers have had similar struggles; the authors of [79] emphasize the difficulties encountered as they tried to modify the 2.4 kernel to use RAM to improve disk I/O performance. While validating ESDC metrics, several Linux kernel bugs were revealed. Since they don't obviously affect kernel functionality, none of these kernel implementation flaws are addressed in any available kernel documentation. Some of the bugs are caused by kernel race conditions that were discovered independently by kernel architects and the author of this thesis. Therefore, the problems are known to the Linux community and future versions of Linux hopefully will be improved. Two kernel race conditions discovered by the author will be summarized below to illustrate the challenges encountered during implementation.

Both race conditions were discovered by examining the consistency of metrics relating to high memory. One analysis involved correlating the number of high memory page allocations with the size of ESDC. Because ESDC is allocated exclusively in the high memory zone, the number of allocated pages in high memory should always be equivalent to the size of ESDC. The number of pages in ESDC, n_{esdc}, can be determined from the page cache size metric in `/proc/meminfo`. The same virtual file also includes metrics for the number of free pages in high memory (n_{free}) and the total number of high memory pages (n_{high}). Therefore, the following equality should always be true:

$$n_{esdc} = n_{high} - n_{free} \,. \tag{4.9}$$

If the left-hand side is greater, then an ESDC overflow exists where the data structures associated with the page cache are claiming more pages than are actually allocated in high memory. When ESDC was implemented, ESDC overflows and underflows were detected for several reasons. First, an incomplete implementation of a function responsible for freeing page cache memory caused most of the ESDC overflows. Second, a race condition in the functions implementing the `truncate()` system call produces an ESDC underflow (the left-hand side of Equation (4.9) is less than the right). In this case, the page cache data structures report fewer pages than are actually allocated in high memory. More information regarding these kernel bugs is available in Section 5.4.3.

4.12 Conclusion

The extended storage disk cache was designed as an adaptation of an existing operating system disk cache. An alternative design would be to create a second disk cache below the existing virtual disk cache in the memory hierarchy. This would have the advantage of minimizing the impact of the reduced performance of extended storage memory. However, the current ESDC design is suitable as a preliminary investigation of the cost-effectiveness of file memory. As well, an ESDC implementation is remarkably straightforward when based on a memory-based disk cache implementation. The stability of ESDC design ensures the success of experiments that apply extreme memory pressure for lengthy periods of time.

> The overall lesson that can be drawn is that seemingly simple changes can have much more far-reaching effects than first anticipated.
>
> — A. Wang, 2002
> *Regarding Linux modifications*
> *for the Conquest file system* [79]

Chapter 5

Experiments

5.1 Introduction

File memory is assumed to have poorer performance than conventional DRAM because several DRAM design constraints may be relaxed to make file memory less expensive. That is, file memory access times may be slower than conventional DRAM by a certain factor, which would depend on the design and implementation of a possibly discontiguous file memory device. To quantify the cost-effectiveness of file memory, the empirical results help to predict the quantity of additional file memory required to achieve equivalent performance. Some results also show the effect that various file memory access times have on performance.

An empirical methodology is used to evaluate ESDC with an emphasis on reproducibility. That is, by accurately specifying the experimental platform and automating the the execution and data acquisition of all experiments, the results presented in this thesis could be reproduced independently. By identifying the sources of experimental error and verifying the accuracy of all ESDC metrics, realistic interpretations of the empirical results are possible. An empirical evaluation of ESDC is based on the results of a variety of experiments. The experimental results are not limited to performance metrics such as miss rates, but also include actual I/O throughput measurements and execution time results. Some of the experiments will evaluate different aspects of ESDC design, such as file memory performance as a general-purpose extended storage disk cache or its impact on demand paging.

5.2 Experimental Methodology

An empirical framework is used to evaluate the impact of ESDC on overall system performance and its cost-effectiveness. A description of this framework involves a specification of the experimental platform and an overview of the utilities created for experimental automation. Finally, a proposal for a new memory hierarchy can be evaluated by different techniques, so a short discussion of alternative models is required.

5.2.1 Cost-Effectiveness Evaluation

File memory is employed as extended storage in an attempt to improve overall system performance. The empirical results can help to determine the minimum quantity of file memory that will begin to offer superior performance without increasing the cost of the system. This relies on the assumption that file memory is slower but less expensive than conventional DRAM. Performance can be improved by simply increasing system memory capacity so that the virtual disk caches in main memory capture larger working sets. This is equivalent to using conventional DRAM as extended storage. If file memory instead of DRAM is used as extended storage, ESDC will have to be larger to increase the ESDC hit rate and obtain the same level of performance. Both alternatives are achievable at the same cost since file memory is substantially less expensive per bit than conventional DRAM.

There are two relationships that can be drawn from the results of the experiments. First, given a capacity of DRAM-based extended storage, the performance results of both alternatives will indicate the minimum amount of additional file memory required to achieve equivalent performance. Second, one can predict the potential performance improvement of using file memory as extended storage, assuming a particular quantity of file memory is available at a price equivalent to a smaller quantity of DRAM. Examples of such relationships can be found in analyses of the results of the PostMark benchmark in Section 5.4.2 and the Bonnie benchmark in Section 5.4.3.3.

Table 5.1: Experimental Platform

Component	Specification
Processor	2.4-GHz Intel Pentium 4
L1 Cache	12-KB instruction; 8-KB data cache
L2 Cache	512-KB unified cache
Memory	2-GB Platinum DDR PC2100 SDRAM 266 MHz
Hard Disk	18-GB SEAGATE Model ST318405LW Rev 0105
Controller	Adaptec 29160N Ultra160 SCSI adapter

5.2.2 Experimental Platform

The use of file memory as extended storage is intended for modern computer systems such as desktop workstations or server applications. Therefore, the system dedicated for ESDC evaluation is a typical personal computer with an Intel Pentium 4 architecture. The specifications of this system, shown in Table 5.1, indicate that all components are typical for a desktop system with two exceptions. First, a total of 2 GB of DRAM is installed to permit configuring experiments with large main memory and extended storage sizes. Second, a reliable SCSI hard disk was selected to avoid disk failure due to the frequent use of I/O-intensive experiments.

The operating system installed on the experimental platform is Slackware Linux 8.1. This distribution features Linux kernel version 2.4.18 [70], a popular kernel that was documented in detail [8] and used for testing compressed caching [10]. A disadvantage of selecting this kernel was the scarcity of documentation during the first year of the ESDC research project. Our 2.4.18 kernel patch for ESDC is shown in Appendix B.

5.2.3 Experimental Automation

To promote experimental accuracy and reproducibility, it is necessary to automate the execution of all ESDC benchmarks and workloads. Every ESDC experiment is specified by a custom configuration file (see Section C.2). A standard configuration file syntax is compatible for all benchmarks, because the command line of the benchmark is specified in the configuration file. Up to two independent parameters can be varied at discrete intervals, which implies that two-dimensional plots or surface plots are supported. These independent parameters might configure properties of the operating system memory usage or might adjust benchmark configuration options.

The configuration file used for benchmarks and workloads governs the execution of a custom Perl script, `mando.pl`. Listed in Section C.2, this script is responsible for configuration file parsing, specification of operating system properties, raw data acquisition, experimental execution and automatic system reboot.[1] After a set of one or more experiments for a particular configuration, the system is rebooted before the next set of experiments is run to avoid hot cache effects [10]. The `mando.pl` script is responsible for managing execution passes and cycles. An *execution pass* refers to all experiments run between system startup and shutdown, while a single experiment is run during an *execution cycle*. Depending on configuration file specifications, the script may activate one or more swap devices, disable file memory caching using the MTRR registers, or set various other ESDC properties. After all initialization has been completed, the experiment is launched. Prior to system reboot, a small startup script is modified with updated command line parameters that will be supplied to the subsequent execution of `mando.pl`. This process is repeated until reaching the specified limits on the number of execution passes.

Due to the volume of empirical data, it is necessary to present the raw data collected during the experiments in multi-dimensional graphs. A general-purpose utility, `dispono.pl`, was created to parse the raw data of each experiment and present the results as two-dimensional graphs or surface plots (see Section C.3). It generates a data file in a format suitable for GNUplot and then generates the plot itself. Cross-sections of surface plots can be presented as a line graph with error bars representing the 95% confidence intervals, which are calculated from the mean and standard deviation of the sets of experiments for each data point.

---

[1] Perhaps it would sound better to write *automatic system rebootion*, but coining such terms is beyond the scope of this thesis.

5.2.4 Alternative Models

Extended storage evaluation could involve the simulation of a software model. For example, extended storage could be evaluated by a modified cache hierarchy simulator or by a customized computer system simulator [84]. However, the experimental error introduced would be significant. Trace-driven simulations have been used to effectively evaluate memory and storage hierarchies in the past [1, 28, 62]. However, it is difficult to collect meaningful traces for extended storage designs [45]. Cache simulators and cache traces are designed for typical cache sizes that are orders of magnitude smaller than system memory sizes. However, extended storage capacities are comparable to or larger than most main memory sizes. Therefore, simulations could take weeks to run even if traces could be obtained.

A simulation methodology will have challenges with accurately modeling the effects of a real operating system due to the assumptions and simplifications that often are made by simulators. For example, operating system simulators may oversimplify complex features of a real operating system, such as using a single virtual disk cache for pages backed to disk. Creating an accurate model of disk I/O scheduling in a simulator is a challenge; a production operating system features optimizations, such as read-aheads and buffered writes, that are suitable for a variety of application workloads. One type of simulation model attempts to address some of these issues [1]. That is, traces of block I/O requests are obtained by modifying an operating system and are provided as input to a memory hierarchy simulation model. However, in [1], disk service times are modeled with an exponential distribution rather than basing seek times on a disk's geometry. Instead of using this hybrid approach for ESDC evaluation, the operating system's hierarchy itself was modified. In this way, the results reflect actual disk I/O and operating system behavior rather than the possibly inaccurate characteristics of a simplified model. Another advantage of avoiding a simulation methodology is that a patched kernel is available for the installation of authentic file memory.

5.3 Experimental Validation

Before the empirical results can be analyzed, it is essential to identify the known sources of experimental error and to verify the accuracy of all ESDC metrics.

5.3.1 Sources of Experimental Error

The results described in this chapter are obtained from direct measurement of ESDC functioning as part of a modified operating system. This approach avoids the most common sources of experimental error that would arise during simulation, including ignoring the effects of the operating system such as read-ahead, write buffering, I/O request scheduling, paging, context switching, and other virtual memory management intricacies. Nevertheless, a number of sources of error exist and are discussed below.

First, modifying the kernel involved introducing minor modifications to various optimized memory management functions. Most modifications have a negligible effect on system performance, except for those that accumulate ESDC metrics that are accessible via the `proc` file system. Even though performance is reduced by a few percentage points, such metrics are essential for evaluating ESDC performance.

Second, the 2.4.18 Linux kernel suffers from a erroneous race condition where the size metrics of the page cache are incorrectly calculated. This can occur when pages are added to the page cache while the `truncate()` system call removes pages. The result is that the size of the available area in ESDC is less than the actual amount of space available, which can reduce temporarily benchmark performance. For most cases, the error introduced in the performance results is only a few percent. However, when a benchmark removes excessively large files and its working set fits entirely within the ESDC, the error can be significant. This occurs only during the Bonnie benchmark, which is described in Section 5.4.3.

Third, disabling processor caches is not possible for every experiment. Some experiments, such as PostMark and Bonnie, function moderately slower with high memory caching disabled, while others, such as kernel compilation, require two or-

Table 5.2: ESDC Read Test

Metric	Before	After	Delta
IO bounce reads	2311	2488	177
IO bounce writes	501	501	0
ESDC reads	5526	5711	185
ESDC writes	1805	1805	0
Number pages freed	32940	32940	0
Number pages allocated	3716	2893	177

ders of magnitude longer execution times. In the former case, the data belonging to mapped pages is not extensively referenced, while the latter situation demonstrates how caching frequently used data lines in SRAM memory provides substantial performance benefits. The excessively long execution times caused by disabling the processor caches for some experiments masks the effect that would be expected from different file memory access times. Therefore, accurate file memory access time ratios are possible for some experiments while others require access time ratio estimation.

5.3.2 ESDC Metrics Verification

As discussed in Section 4.10, a variety of ESDC metrics are available to monitor ESDC behavior. The number of metrics available permits making correlations among several metrics that monitor a given ESDC property using different methods. ESDC metrics are based on monitoring page transfers to and from high memory. To verify that the understanding of the associated mechanisms are correct, several simple experiments are analyzed. These experiments are performed with abundant ESDC memory and conventional DRAM memory.

The first experiment confirms that ESDC functions correctly when reading a file on disk. First, initial ESDC metrics are recorded by using `cat` to access `/proc/esdc`. Then, the standard word count utility, wc, processes a 708124 byte file. The kernel should map at least 172 pages and then transfer these through the

Table 5.3: ESDC Read and Write Test

Metric	Before	After	Delta
IO bounce reads	3091	3238	147
IO bounce writes	189	189	0
ESDC reads	6816	7126	310
ESDC writes	1017	1297	280
Number pages truncated	8	148	140
Number pages freed	32776	32916	140
Number pages allocated	3110	3537	427

ESDC stage in the memory hierarchy. An additional 5 allocated pages are needed for loading the `wc` utility. Therefore, 177 pages need to be read from disk. As shown in Table 5.2, a total of 177 buffer pages pass through the bounce buffers. As well, 177 pages are allocated in ESDC in high memory. An extra 7 pages in addition to the 177 pages are transferred between high memory and user space for ESDC hits caused by executing the second `cat /proc/esdc` command. Thus, this experiment accounts for all ESDC pages involved with reading regular files from disk.

Another experiment permits an analysis of more ESDC metrics. The following command reads a file from disk, pipes the standard output to sort and then writes the output to a new file on disk:

```
% cat foo | sort > bar
```

The file `foo` is a 571072 byte file of random data that would require 140 pages to be allocated in ESDC memory. Before the experiment, the `cat` utility already is cached in ESDC memory. However, the 26632-byte `sort` utility has to be read from disk into 7 pages of memory. Therefore, 147 pages are transferred from the bounce buffers into ESDC memory, as shown in Table 5.3. During this experiment, the 140 pages associated with the file `foo` are processed and replicated twice because an extra 280 pages are written to ESDC memory. These 280 pages plus the original 147 pages required for disk reads implies that 427 pages are allocated in the

Table 5.4: Experimental Suite

Type	Name	Version	Application
Synthetic benchmark	Postmark	1.5	Small file I/O [39]
Synthetic benchmark	Bonnie	1996	I/O performance [9]
Application workload	MUMmer	3.10	Genome alignment [18]
Application workload	Kernel compiling	2.4.18	Linux kernel build [10]

high memory zone. One ESDC metric indicates that the `truncate` system call is used to truncate 140 pages. This shows that pages associated with an intermediate file are removed from ESDC instead of from disk. This is an example of how a disk cache like ESDC reduces the number of disk I/O operations.

5.4 ESDC Experiments

5.4.1 Experimental Suite

In this thesis, an *experiment* is understood to be either a synthetic benchmark or an application workload. A *synthetic benchmark* attempts to emulate the behavior of a set of programs by an artificial algorithm. An *application workload*, however, is a real program that can be used to evaluate performance while it performs a useful task [25]. All benchmarks and applications are available in the public domain, which promotes reproducibility of the experimental results. These programs are executed without modification, with most of the configuration parameters set to the default values. For each experiment, however, several key configuration parameters are set to create large working sets that force memory pressure. Most working sets are approximately 80 MB in size.

ESDC has been evaluated using a variety of benchmarks and workloads. Table 5.4 lists the components of an experimental suite that force different patterns of I/O and memory usage to exercise the I/O subsystem and the virtual memory management of the modified Linux kernel. The results produced each by these experiments will be discussed in detail in the following sections.

5.4.2 PostMark Synthetic Benchmark Results

Created in 1997, PostMark [39] is a single-threaded synthetic file system benchmark. Instead of using workloads based on a few large files, PostMark simulates heavy system loads that are dominated by frequent accesses to many short-lived small files. That is, PostMark is designed to measure the performance of electronic mail servers, news servers, Internet service providers, and workstations such as those running large engineering design applications that manipulate thousands of small files.

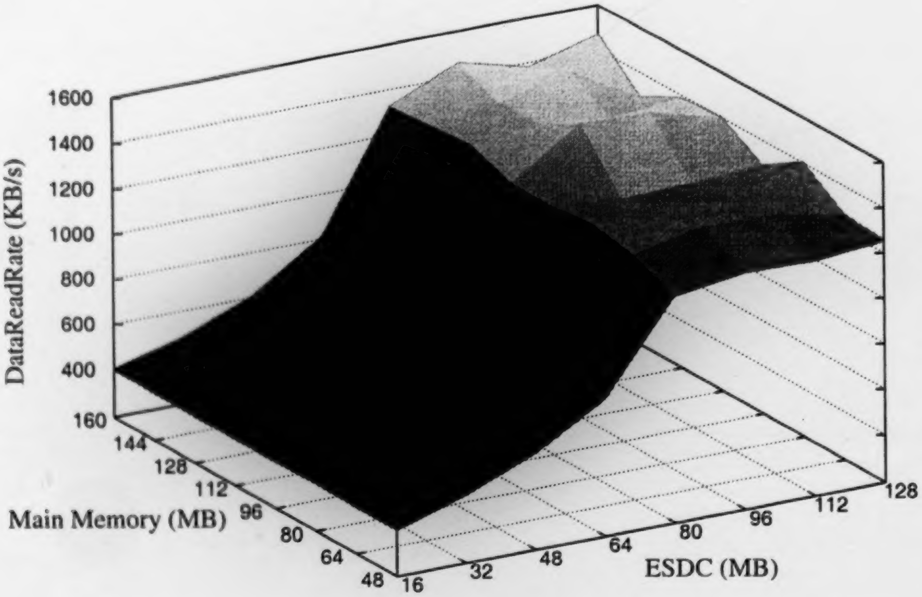
5.4.2.1 Baseline performance

Before the results of the first PostMark experiment can be discussed, the performance of this benchmark under several reference conditions must be analyzed. First, if ESDC consisted of conventional DRAM, it would function with access times equivalent to main memory. The performance of PostMark with ESDC as DRAM for read and write operations is shown in Figures 5.1(a) and 5.1(b), respectively. Each of the sixty four data points on these graphs represents five execution cycles of the PostMark benchmark, where each cycle creates 10000 small files, with sizes ranging from 500 bytes to 10004 bytes. A total of 10000 I/O transactions are performed on these files. The system is restarted between each data point. All PostMark experiments are run with 128 MB of available swap space. When paging was necessary, only a small percentage of available swap space was used.

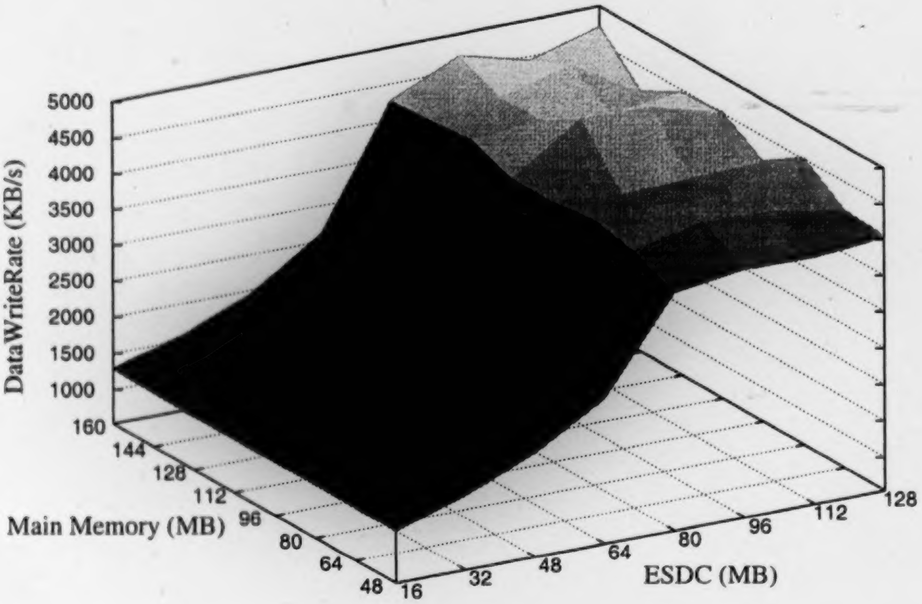
The observed performance is considered typical of a 32-bit Linux system that makes use of high memory [8]. However, as discussed in Section 4.8, the accuracy of ESDC access time penalties would be adversely affected by caching of high memory pages. Therefore, to establish a baseline measurement of performance, caching must be disabled for the high memory zone utilized by ESDC.[2] The performance of this reference case for reads and writes is shown in Figures 5.2(a) and 5.3(a), respectively. For large ESDC sizes, the peak performance of the system us-

---

[2] Note that this would be done only for purposes of evaluating ESDC. An actual ESDC-equipped system would not require ESDC caching to be disabled.



(a) PostMark read performance.



(b) PostMark write performance.

Figure 5.1: PostMark results with cached DRAM as ESDC (no access time penalty).

ing uncached high memory is about 60% of the peak performance of cached high memory.

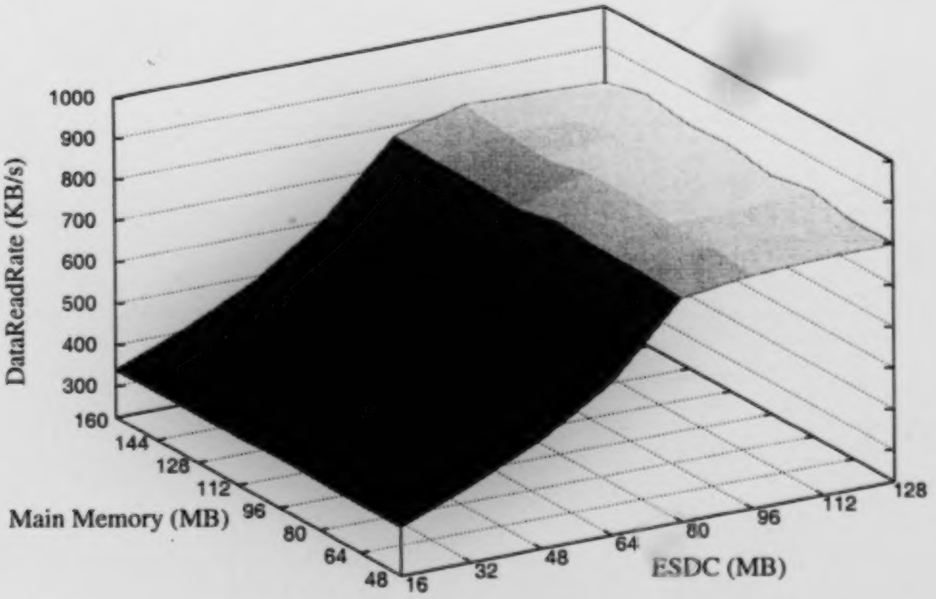
5.4.2.2 PostMark using File Memory

Now that the reference performance of PostMark has been obtained, the performance of file memory as ESDC can be evaluated. The key question is the extent of the reduction in PostMark performance when ESDC uses file memory with slower access times than conventional DRAM. Our default normalized access time ratio is 3, which is the ratio of file memory access time to DRAM access time. This configuration produces the PostMark performance results shown in Figures 5.2(b) and 5.3(b). Before analyzing these results, it is informative to emphasize some observations. First, the size of main memory appears to be irrelevant in this situation; PostMark performance clearly depends on the size of extended storage. The plateaus that form after about 80 MB are symptomatic of the working set size. That is, the experiments involved 10000 files with sizes up to 10000 bytes, so the maximum possible data set size would be less than 100 MB.

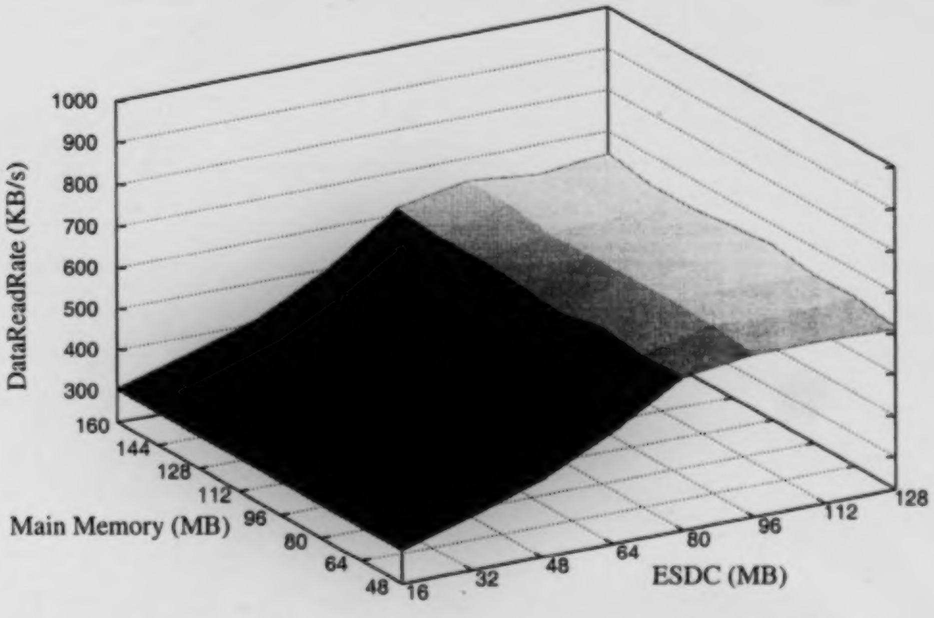
5.4.2.3 Analysis of PostMark Performance

While surface plots can help to illustrate performance trends, line graphs are necessary to compare the performance of file memory with conventional DRAM. Vertical cross-sections of the surface plots in Figures 5.2 and 5.3 are presented in the graphs in Figures 5.4 and 5.5 for a constant main memory size of 112 MB. The error bars represent the 95% confidence intervals calculated from the results of the five execution cycles that were run for each data point.

Figures 5.4 and 5.5 show the performance of ESDC using file memory that is three times slower than conventional DRAM. At first glance, the 22% reduction in peak performance may suggest that file memory is not effective, but this conclusion ignores cost considerations. A key feature of file memory is that it is substantially more economical than conventional DRAM. Supposing that chips based on multilevel DRAM technology store two bits per cell, then file memory is half of the

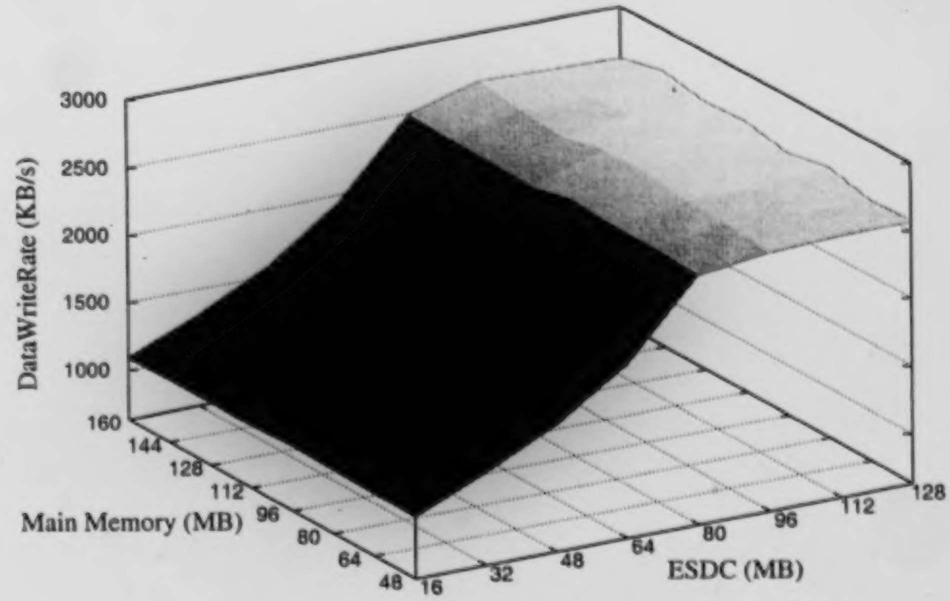


(a) ESDC read performance using uncached DRAM with no access time penalty.

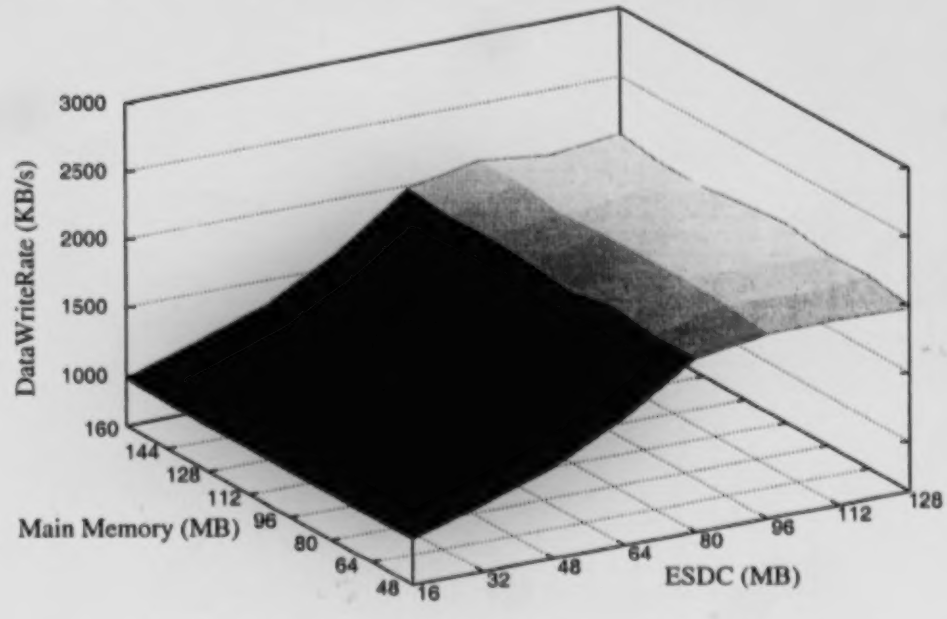


(b) ESDC read performance using file memory with access time ratio of 3.

Figure 5.2: PostMark read rate comparing file memory as ESDC with DRAM as ESDC.



(a) ESDC write performance using uncached DRAM with no access time penalty.



(b) ESDC write performance using file memory with access time ratio of 3.

Figure 5.3: PostMark write rate comparing file memory as ESDC with DRAM as ESDC.

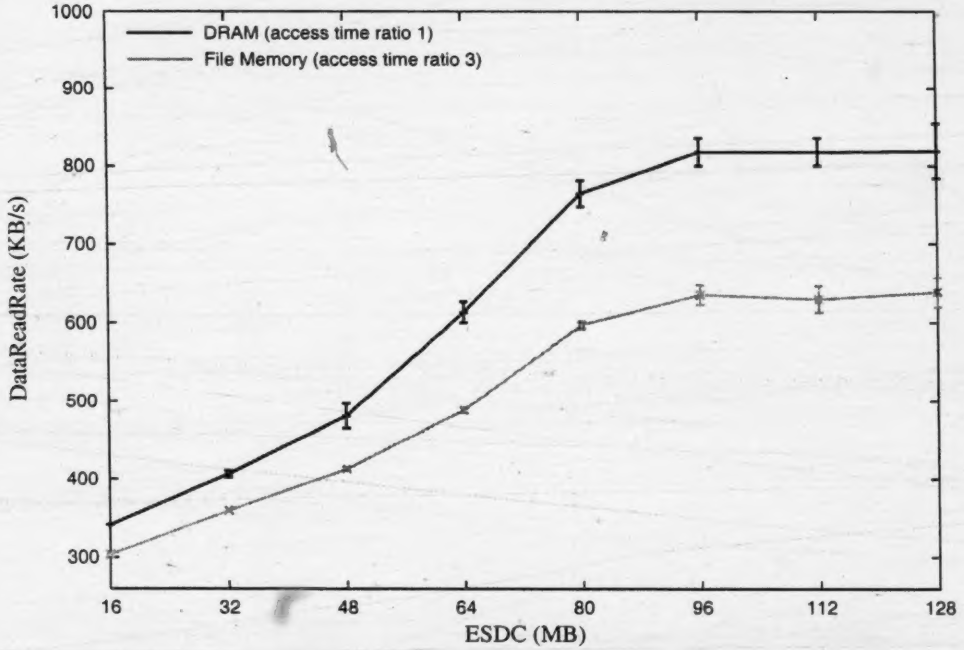


Figure 5.4: PostMark read rate with main memory fixed at 112 MB and a file memory access time ratio of 3. Error bars represent 95% confidence intervals.

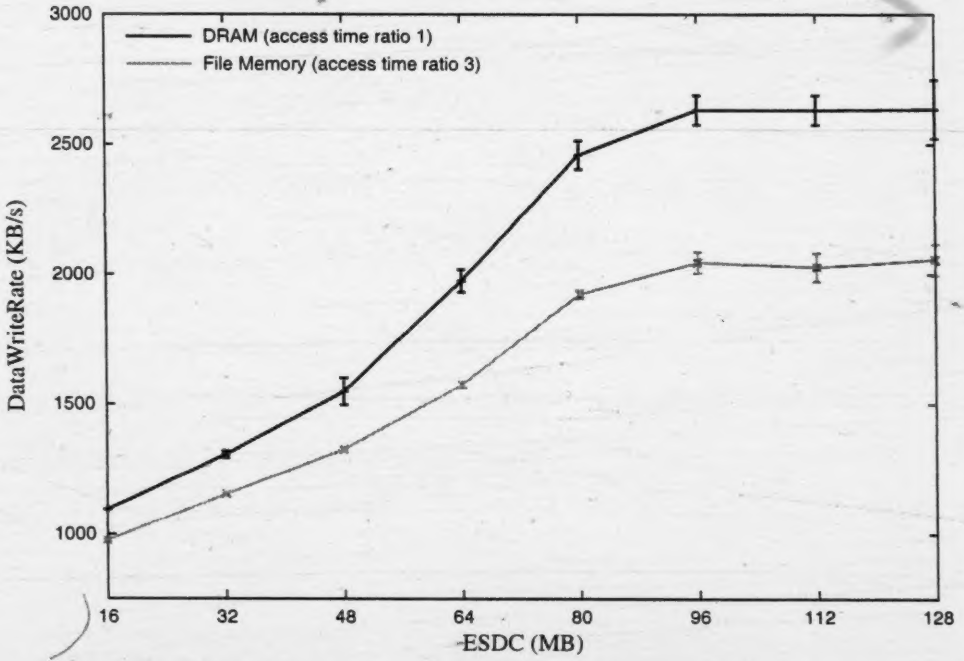


Figure 5.5: PostMark write rate with main memory fixed at 112 MB and a file memory access time ratio of 3. Error bars represent 95% confidence intervals.

cost of conventional DRAM. This implies that the capacity of extended storage using file memory would be twice the quantity of the DRAM available for the same purposes on a conventional system, assuming a constant overall budget. Figures 5.4 and 5.5 indicate that performance improvements are possible for systems with limited memory capacities. For example, 40 MB of conventional DRAM would offer 440 KB/s for PostMark reads and 1425 KB/s for PostMark writes. However, since 80 MB of file memory could be purchased at the same cost, read performance would increase to 600 KB/s and 1925 KB/s for PostMark reads and writes, respectively. Therefore, PostMark shows that the use of substantially slower file memory as ESDC can offer 36% and 35% better performance for reads and writes, respectively, without increasing costs.

Given a DRAM-based ESDC, another question is the capacity of file memory required for equivalent performance. From Figures 5.4 and 5.5, if it were to be possible to add 62.5 MB of DRAM to a system, 80 MB of file memory would be needed as ESDC for equivalent performance. That is, only 28% more file memory than conventional DRAM is necessary for equivalent performance. If three similar measurements are made over the working set range (16 MB to 80 MB), the average additional file memory is 37%. PostMark results indicate that, if file memory can be at least 27% cheaper than conventional DRAM while suffering a performance loss of a factor of three, it can be used as extended storage to increase the performance of a system.

To quantify the cost-effectiveness of file memory, the PostMark benchmark can be used to determine the effect different file memory access times have on performance. Therefore, in another PostMark experiment, a range of access time penalties are specified for different ESDC sizes. Since PostMark performance is essentially independent of the main memory size, main memory was set at a constant value of 160 MB. The performance results are presented as surface plots in Figure 5.6. File memory access time ratios are normalized to conventional DRAM, which has an access time ratio of 1. This reference access time is expressed in the graphs as 100%.

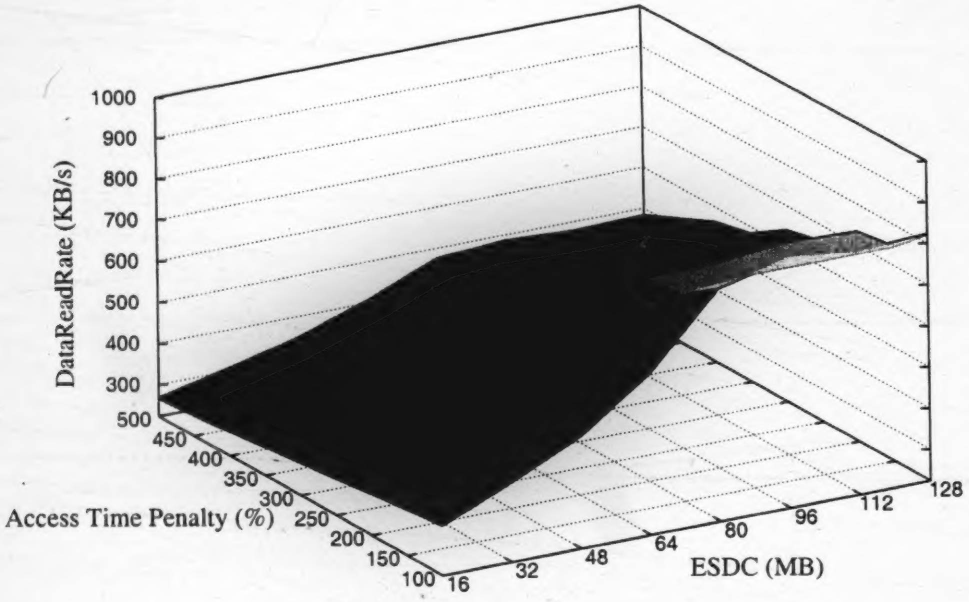
Table 5.5: Impact of ESDC with File Memory Access Time Ratio of 2

PostMark Operation	**ESDC Size (MB)**	**ESDC as DRAM (KB/s)**	**ESDC as FM (KB/s)**	**Performance Reduction (%)**
Read	64 MB	615.9	554.8	9.9
Write	64 MB	1982.3	1785.6	9.9
Read	112 MB	812.9	735.9	9.5
Write	112 MB	2616.5	2368.8	9.5

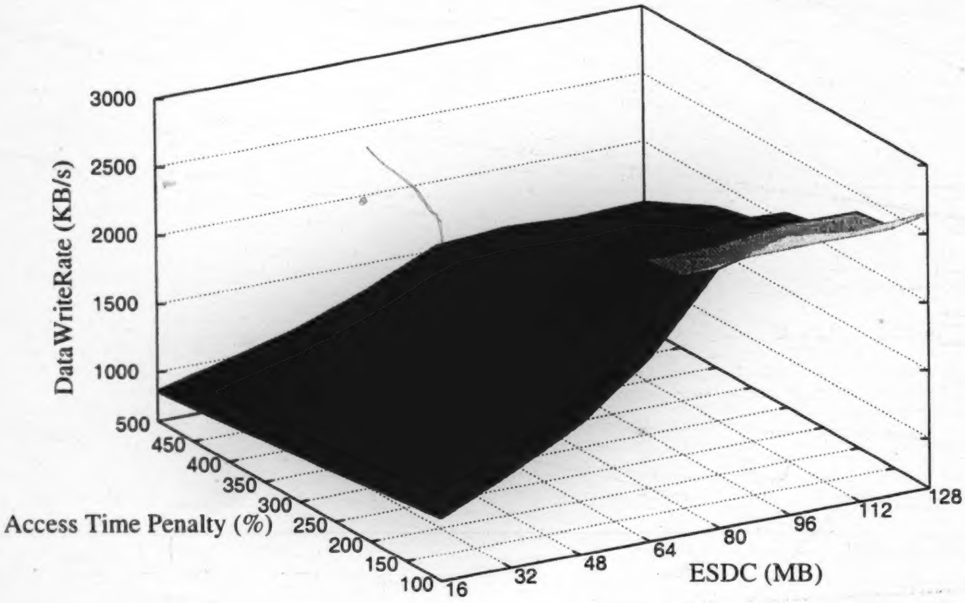
Generally, PostMark performance degrades linearly for increasing file memory access times. At first, it appears that the rate of performance degradation is more pronounced for large ESDC sizes. However, the slower file memory has the same relative effect on performance regardless of ESDC size. The relationships can be seen more clearly in Figure 5.7, which is a cross-section of Figure 5.6 for a constant ESDC size of 64 MB. Even if file memory is twice as slow as DRAM (access time penalty of 200%), the read and write performance of the PostMark benchmark only drops by 9.9% for both types of I/O. The mean data rates used to calculate this rate of performance reduction are shown in Table 5.5. To determine if this relationship is true for larger extended storage capacities, a second cross-section is presented in Figure 5.8 for 112 MB of ESDC. Since more extended storage is available, performance is consistently higher than in Figure 5.7. It is interesting to note that if file memory with an access time ratio of 2 is used for extended storage instead of DRAM, both the read and write data rates are reduced by 9.5%. Therefore, the impact that the file memory access time has on the PostMark benchmark is independent of the size of extended storage and the I/O data rate.

5.4.2.4 Analysis of PostMark Miss Rates

Another method of evaluating ESDC is to measure the miss rate during the execution of a benchmark. This was accomplished by a script that periodically polls the set of ESDC metrics stored in `/proc/esdc`. The ESDC miss rate, expressed as a percentage, is one of these real-time metrics. The modified kernel calculates



(a) ESDC read performance for various file memory access times.



(b) ESDC write performance for various file memory access times.

Figure 5.6: PostMark performance versus ESDC and file memory access times (shown as a percentage of DRAM access time). Main memory is fixed at 160 MB.

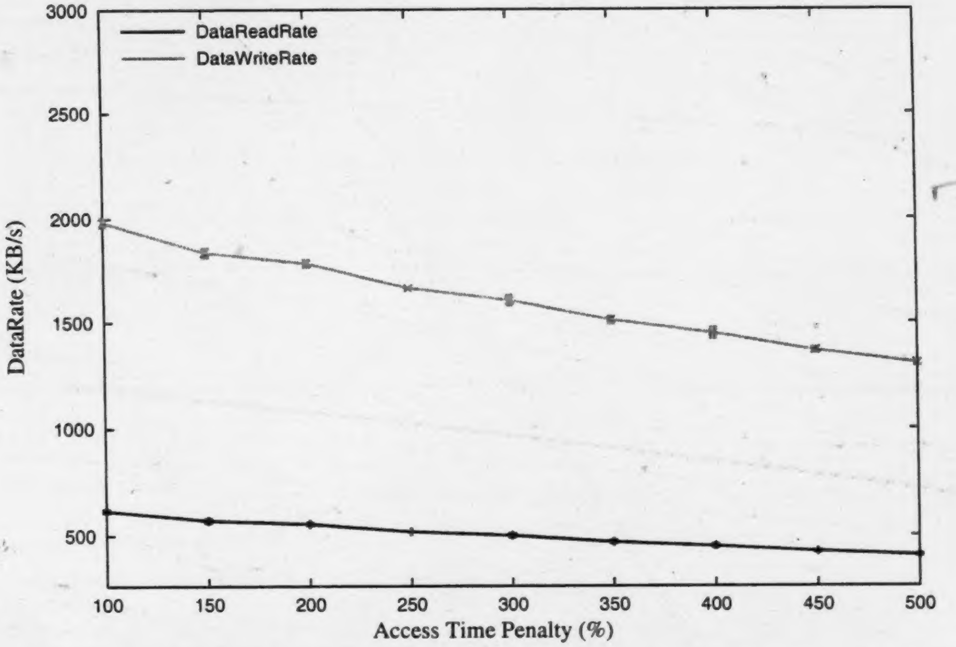


Figure 5.7: PostMark performance with main memory fixed at 160 MB and a constant ESDC size of 64 MB. Error bars represent 95% confidence intervals.

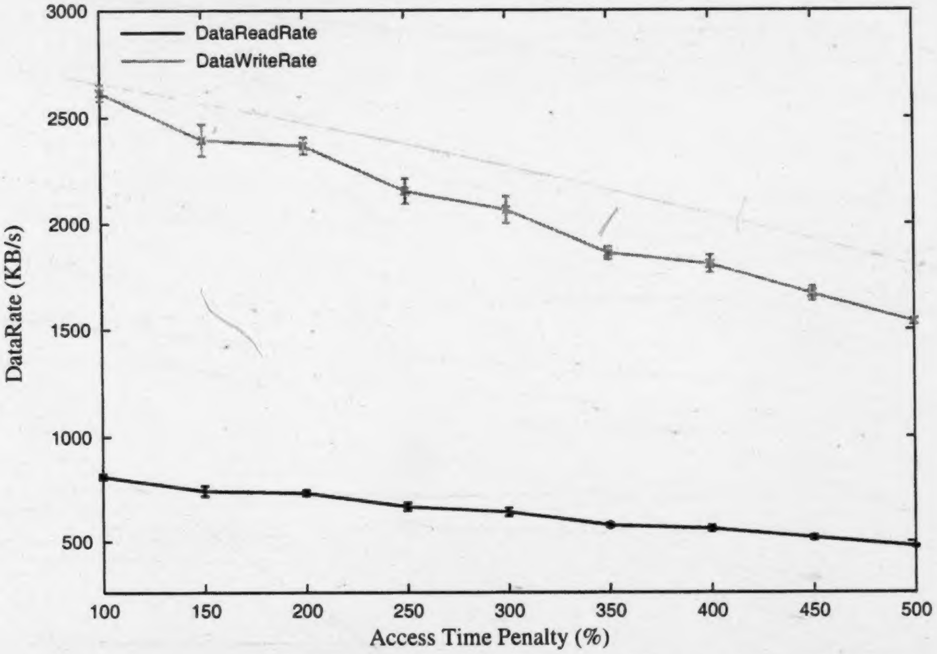


Figure 5.8: PostMark performance with main memory fixed at 160 MB and a constant ESDC size of 112 MB. Error bars represent 95% confidence intervals.

the current miss rate from the cumulative totals of hits and misses. A miss occurs whenever a requested page is not found in ESDC and is instead fetched from disk.

The ESDC miss rate was monitored while executing the PostMark benchmark for two different memory configurations. The monitoring spanned five consecutive execution cycles of the benchmark, which is equivalent to the number of cycles used to determine the mean for each data point of the PostMark performance plots discussed earlier. A plot of ESDC miss rates for the first memory configuration (160 MB of main memory and 16 MB of ESDC) is shown in Figure 5.9. The miss rate rises quickly and then gradually levels off, asymptotically approaching 92%. When PostMark is executed on a system with large main memory capacities but small ESDC sizes, the miss rate of ESDC is very high. This occurs because ESDC is too small to adequately cache the pages involved with I/O operations. Since relatively large numbers of transactions are performed on many small files, the working set is much larger than ESDC and subsequent accesses for a mapped page would not occur before it is replaced with a more recently mapped page.

A second memory configuration preserves the main memory capacity as 160 MB, but increases the size of ESDC to 128 MB. ESDC metrics were monitored in the same way as above, and the results are shown in Figure 5.10. As before, the miss rate starts off low, but increases as the experiment proceeds. However, the miss rate levels off at 63%, which is much lower than the 92% peak when there is only 16 MB of ESDC. Clearly, the additional ESDC capacity reduces the miss rate as well as increases the performance of the benchmark discussed earlier. Further increases in ESDC capacity do not affect the miss rate due to the I/O access patterns created by the PostMark benchmark algorithm. In fact, the miss rate remain at 63% even for 512 MB of ESDC and 160 MB of main memory. The steps visible in Figure 5.10 are artifacts of the five consecutive executions of the benchmark. Each additional benchmark instance increases the miss rate slightly as a new working set replaces the previous set of cached pages.

The observed behavior of PostMark shows how adding extended storage can reduce the miss rate of a virtual disk cache and improve the performance without

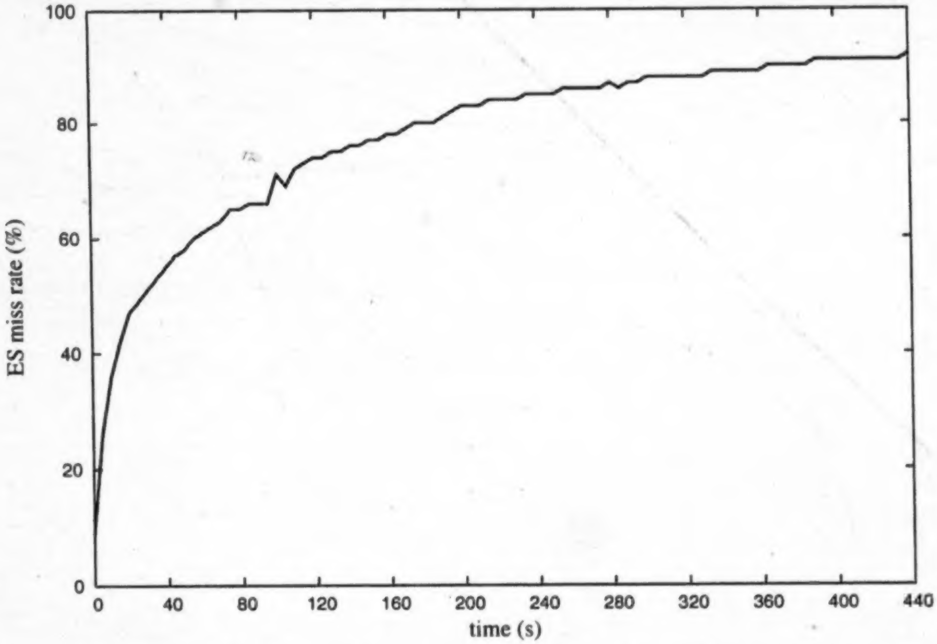


Figure 5.9: ESDC miss rate while running PostMark with 16 MB of ESDC and 160 MB of main memory.

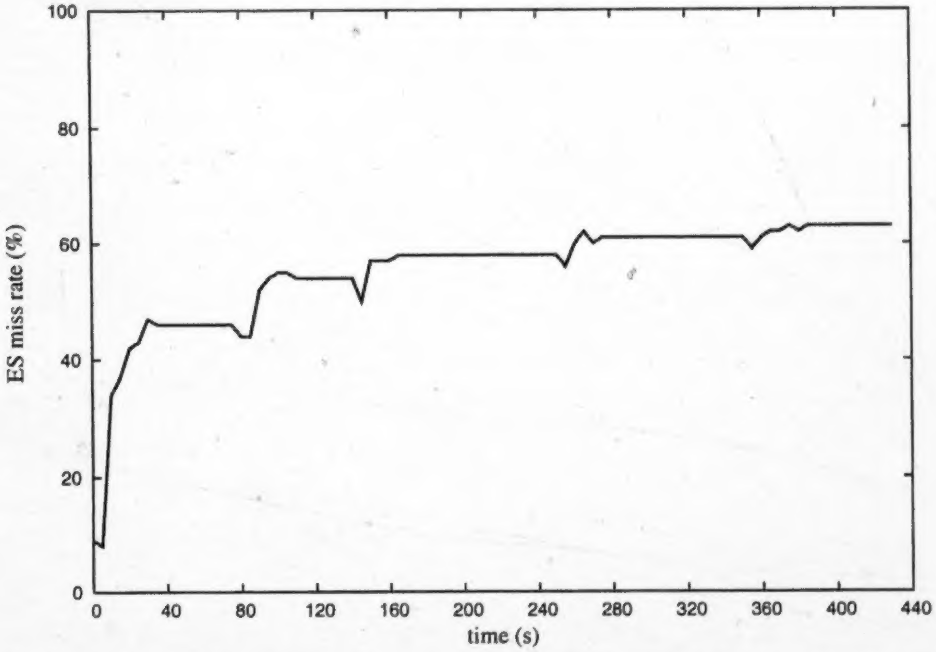


Figure 5.10: ESDC miss rate while running PostMark with 128 MB of ESDC and 160 MB of main memory.

impacting costs. These results demonstrate that ESDC is effective at improving the performance of inexpensive servers such as engineering file servers or servers used by Internet service providers. Such systems are prone to high system loads due to frequent accesses to large number of small files and would benefit from large and inexpensive extended storage disk caches.

5.4.3 Bonnie Synthetic Benchmark Results

Bonnie is a popular and simple file system performance benchmark [9]. Using a single file of known size, it performs various block and character I/O operations. During a single execution, it performs writes using character I/O transfers, reads the file using character I/O, reads and rewrites it with block I/O, writes the entire file again with block I/O, reads the file with character I/O and then with block I/O, and finally performs random seeks. While Bonnie was intended to be used to evaluate ESDC performance, it became a utility to diagnose two related but separate problems with the implementation of Linux 2.4.18. That is, Bonnie was used to improve the accuracy of ESDC metrics because one of the problems was solved. Unfortunately, Bonnie read and write rates are not suitable to be used in a performance evaluation because they are adversely affected by a second kernel bug present in the official release of Linux 2.4.18. Nevertheless, the results of Bonnie's unrealistic random I/O experiments offer some insight into the worst case performance of ESDC.

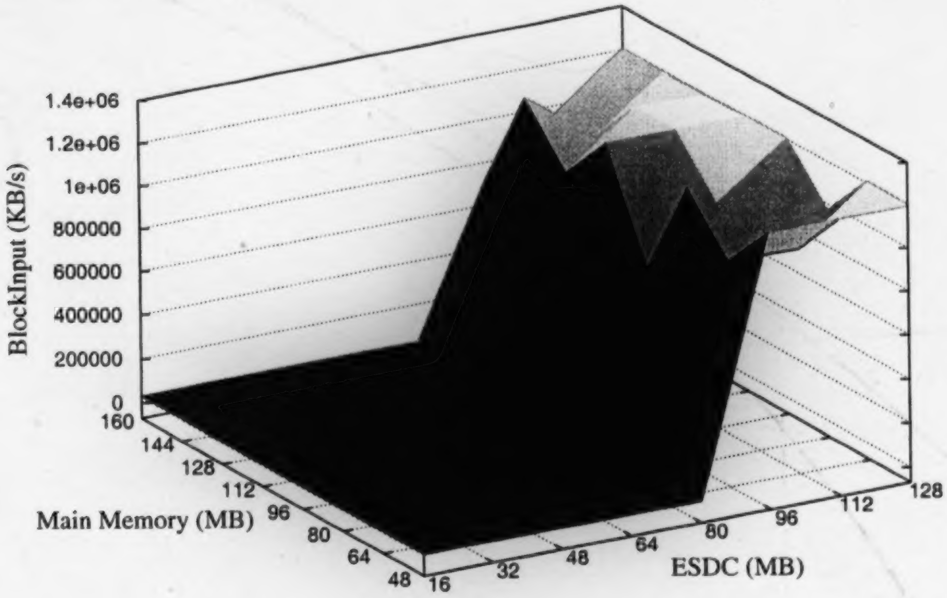
Bonnie was instrumental in isolating two major errors in the official implementation of the 2.4.18 Linux kernel. Although obscured from view in the official kernel, these bugs were revealed by inconsistencies in the ESDC metrics. That is, overflows or underflows occurred when the size of the page cache did not correspond to the amount of pages actually allocated in high memory (see Section 4.11 for more information). The first kernel bug caused frequent page cache and high memory inconsistencies. The resulting overflows accumulate in small increments and cause any ESDC metrics to be unreliable. The solution that I developed for this kernel bug is discussed below. A more complicated kernel flaw only occurs under particular circumstances and only causes underflows. This bug was caused by a situation where a race condition occurs, creating large discrepancies in the reported size of the page cache. Various attempts were made to address this bug, but none were implemented in the final patch due to the risk of introducing instability into the kernel.

5.4.3.1 Balance Classzone Kernel Bug

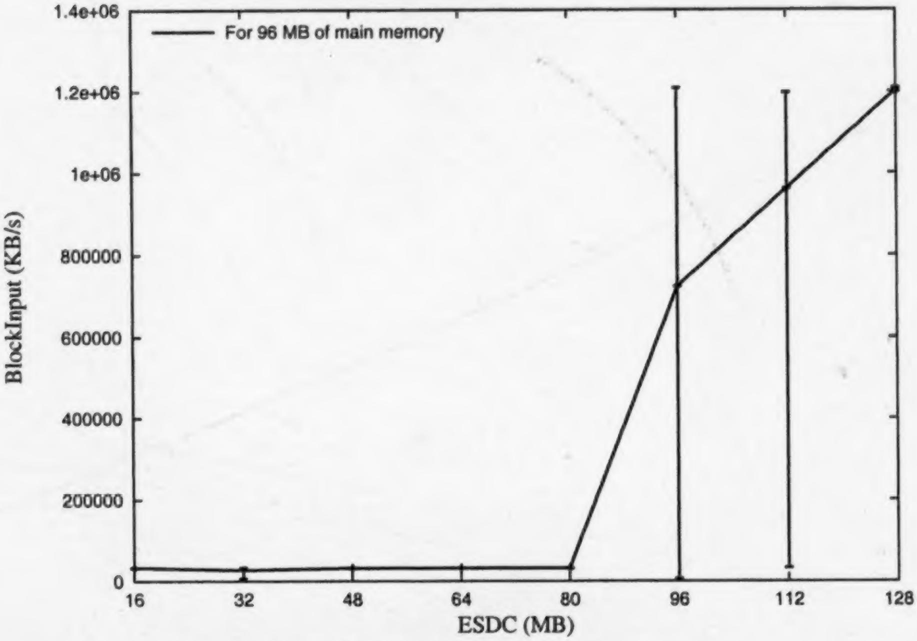
The first ESDC experiments involved simple performance tests where Bonnie performed I/O on an 80-MB file. However, frequent inconsistencies appeared in the results where the reported size of ESDC would overflow the capacity of the high memory zone. One cause of ESDC overflows is a result of an incomplete function in the kernel. This function, `balance_classzones()`, is responsible for freeing memory under situations of extreme memory pressure. However, this function uses local lists of free pages instead of the free page list associated with the current zone. This local list is not needed to avoid failing allocations; it simply helps to provide better fairness so that a task is not freeing memory for the benefit of other tasks rather than itself. In version 2.4.18 of the kernel, this function only maintains a list of one page. Since it also has a side effect of causing an ESDC overflow of one page every time it is called, the solution is to avoid maintaining the local free lists. The pattern of ESDC overflows caused by this kernel bug did not appear after the function was modified to bypass the creation of local free lists.

5.4.3.2 Concurrent Truncate Kernel Bug

When analyzing the results of the Bonnie experiments run on an 80-MB file, a second bug was discovered in the 2.4.18 Linux kernel when the page and swap caches were constrained in high memory. As shown in Figure 5.11(a), the step function rises to a jagged plateau. In other benchmarks, such as PostMark, the plateau is relatively flat. The coombs in the plateau represent points that are the average of a wide range of block data rates. This can be seen clearly in Figure 5.11(b). The large error bars represent minimum and maximum data points instead



(a) Bonnie read performance showing plateau with coombs.



(b) Bonnie read performance for 96 MB of main memory.

Figure 5.11: Visualization of race conditions between the truncate() system call and adding pages to the page cache.

Table 5.6: Analysis of Behavior Caused by Kernel Race Condition

ESDC (MB)	**Main Memory (MB)**							
	160	**144**	**128**	**112**	**96**	**80**	**64**	**48**
128						*		.
112	*	*	.		*	*	?	
96	*		*		*		*	
80	+	s	s	s	s	s	s	s
64	s	s	s	s	s	s	s	+
48	s	s	+	s	s	s	s	s
32	s	S	S	S	+	s	S	s
16	S	S	S	S	S	S	S	S

Key	
S	Many requests to swap processes
s	Few requests to swap processes
+	Some overflows
-	Some underflows
*	Many underflows but no swap requests
.	Some underflows but no swap requests
	Normal behavior
?	Missing logging data in syslog

of confidence intervals.

The source of the problem was identified by examining custom kernel output directed to the Linux system logs. As shown in Table 5.6, the kernel behavior observed in the system log was recorded as a symbol for each data point. There is a direct correspondence between the burst of underflows and the coombs in the plateau. After much investigation, the cause of the underflows was isolated in the kernel and the problem was caused by the implementation of the `truncate()` system call.

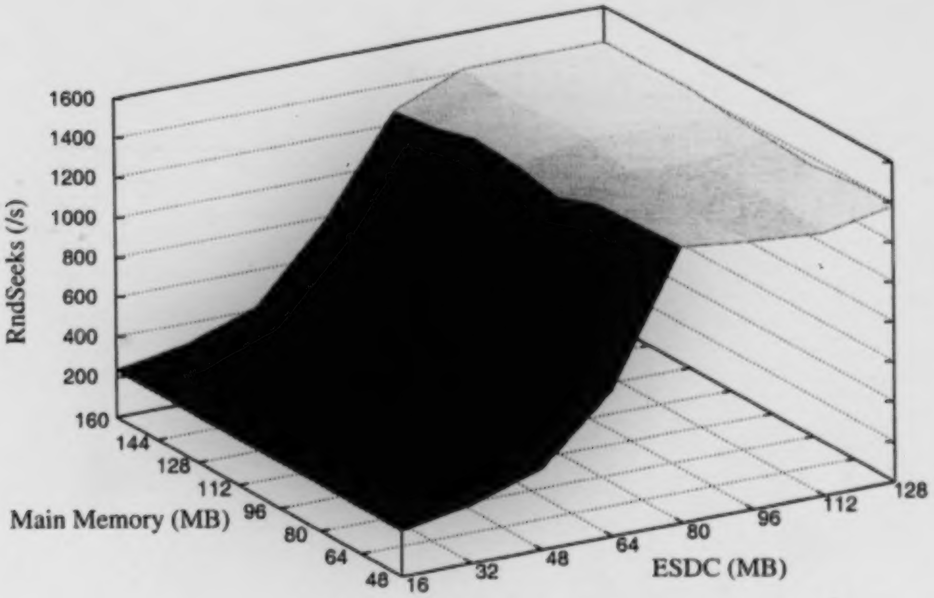
The problem of reduced performance can occur during periods of intense memory pressure when many pages are added to the page cache while the `truncate()`

system call removes pages mapped to large files. The result is that the size of the available area in ESDC is less than the actual amount of space available. Since it appears that less high memory is available, the kernel does not allocate as many pages in ESDC as it otherwise would, temporarily reducing benchmark performance. Most of the time, the inconsistency is either not manifested or is minuscule. However, when a benchmark removes large files and its working set fits entirely within the ESDC, the underflow can involve thousands of pages. If this problem did not occur, the function in Figure 5.11 would appear more uniform. Since the error introduced in most Bonnie results is large, only a partial analysis is possible for evaluating the effectiveness of file memory as ESDC.

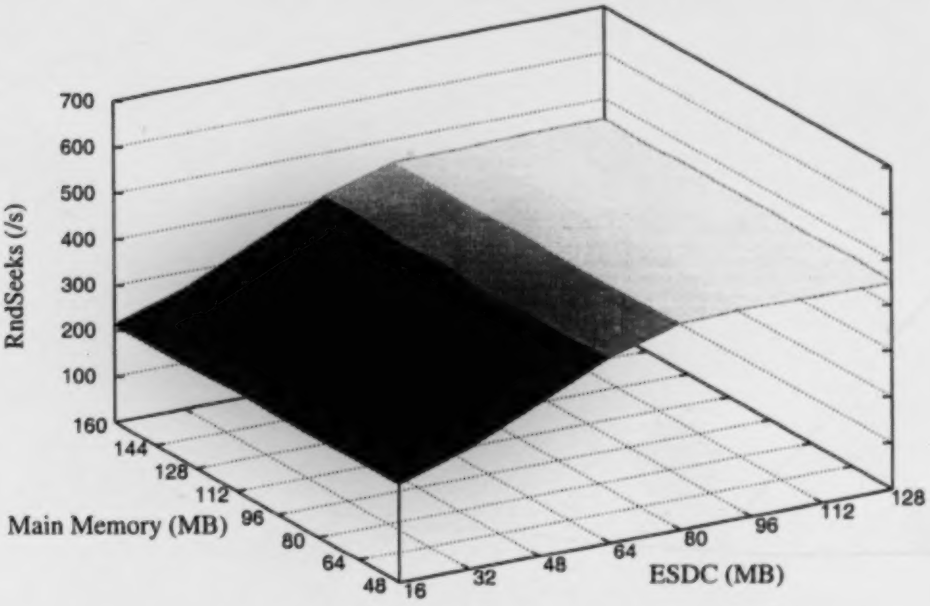
5.4.3.3 Random Seek Performance Analysis

Bonnie random I/O results were not affected by the above issues, which permits an analysis of a worst case performance of ESDC. Figure 5.12 compares the performance of DRAM-based extended storage and ESDC using file memory that is three times slower than conventional DRAM. Unlike the PostMark results, the substantial reduction in peak performance for high ESDC capacities indicates that the current ESDC architecture will suffer from sustained I/O operations on a single large file. The difference in peak performance is approximately a factor of three, which is consistent with both the access time ratio of file memory. A similar peak performance difference is evident for the throughput of block reads and writes.

For systems with limited memory, it is possible to obtain an estimate of file memory cost-effectiveness. For a constant main memory size of 112 MB, a section of the previous surface plots is shown in Figure 5.13. Due to experimental execution durations in excess of 48 hours, only three execution cycles are used for calculating the 95% confidence intervals. If 40 MB of conventional DRAM is available for use as extended storage, Bonnie can perform 350 random seeks per second. Without increasing costs, 80 MB of file memory can offer 450 random seeks per second. Thus, Bonnie shows that using file memory instead of DRAM can offer 29% performance improvement, even with an architecture that does not isolate the access



(a) ESDC random seek rate using uncached DRAM with no access time penalty.



(b) ESDC random seek rate using file memory with access time ratio of 3.

Figure 5.12: Bonnie random seek rate comparing file memory as ESDC with DRAM as ESDC.

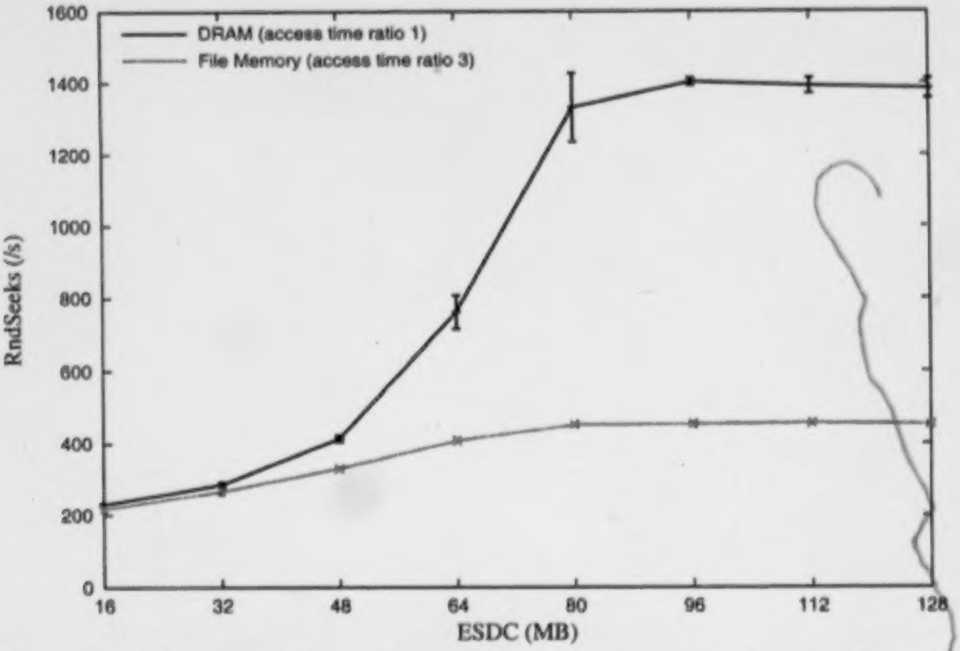


Figure 5.13: Bonnie random seek rate with main memory fixed at 112 MB and a file memory access time ratio of 3. Error bars represent 95% confidence intervals.

time of file memory from the operating system.

The capacity of file memory required for equivalent performance can also be obtained from Figure 5.13. If ESDC consisted of 52 MB of conventional DRAM, then 80 MB of file memory would be necessary to achieve 450 random seeks per second. That is, 54% more file memory than DRAM is necessary for equivalent performance. Since the Bonnie test file is 80 MB in size, a total of four equivalent performance measurements can be made over the range of the working set (80 MB, 64 MB, 48 MB and 32 MB of ESDC). Averaging these measurements indicates that 31% more file memory than DRAM is necessary for equivalent performance. That is, if file memory can be at at least 24% less expensive than DRAM but is three times slower, it can be employed as extended storage to improve the performance of a system. Postmark had similar equivalent performance results, even though PostMark uses thousands of test files while Bonnie performs all of its operations on only one file.

The results of the Bonnie benchmarks indicate that an additional virtual disk cache should be investigated to determine if such an architecture is successful in removing the peak performance gap when using slow file memory. Bonnie documentation, however, indicates that performing random seeks on a file to extract single words is a contrived experiment that is intended to defeat file caching to provide a suitable measurement of disk and file system performance. It is acknowledged that such random file seek operations are less common than page mode accesses.

5.4.4 MUMmer Application Workload Results

MUMmer is an open-source application for the rapid alignment of large DNA and protein sequences [18]. MUMmer is capable of aligning incomplete or entire genomes, but it can require hundreds of megabytes of main memory for large genomes. For example, MUMmer 3.0 can find all 20-basepair or longer exact matches between a pair of 5-megabase genomes using 78 MB of memory. It even is possible to align the entire human genome to itself, but this requires up to 3700 MB of memory. The algorithm finds all maximal unique matches (MUMs) between two input sequences using a suffix tree data structure [19]. Suffix trees can be constructed and searched in linear time using linear space. A “query” sequence is “streamed” past the reference suffix tree so that the memory resources only depend on the size of the reference sequence.

5.4.4.1 Specifications

The MUMmer application requires extensive memory capacities and lengthy execution times to align large genomes. The program will load the sequences and then begin the matching algorithm. As the execution proceeds, the main memory footprint of the process steadily increases until reaching capacity. Then paging to backing store begins if main memory capacities are restricted. Since the majority of the applications pages are backed by swap and the swap cache is utilized extensively, this is a good candidate for evaluating if the swap cache in ESDC can improve demand paging performance.

Table 5.7: Results of the MUMmer Experiments

ESDC Size (MB)	Main Mem (MB)	Elapsed Time (s)	System Time (s)	Total Major Faults	Total Minor Faults	Total Laundered pages	Bounce Reads (pages)
256	384	1988	4.14	22249	97178	138171	25960
256	320	2071	18.28	28724	267990	3241240	41544
256	256	2249	55.87	41211	354241	5387858	57897
192	384	1988	4.10	22362	107716	1196968	26615
192	320	2041	14.66	26449	185117	1380774	40789
192	256	2215	50.71	38790	369588	5498474	57730
128	384	1994	4.21	22225	96324	20347	25770
128	320	2052	14.87	27504	198482	1123785	40516
128	256	*no data due to thrashing*					
160	288	2110	29.44	32593	277154	3491002	48436
160	272	2164	32.23	35002	303243	3548277	53214
144	288	2131	26.20	34100	280795	2863806	49219
144	272	2172	30.06	36909	300842	3557439	54199
128	288	2167	25.28	38918	288434	4079183	59337
128	272	*no data due to thrashing*					

To establish memory pressure for large ESDC and main memory capacities, two chromosomes from the recently published human genome are selected as input sequences. Both are available in the FASTA sequence format with ASCII characters representing the different nucleic acid codes. For all experiments, the reference sequence is the second human chromosome (68.4 MB in size) and the query sequence is the twenty-second human chromosome (9.76 MB). When MUMmer loads these compressed sequence files, the total allocated space in ESDC increases to 80 MB to handle the associated file-backed pages. Most of the remaining space in ESDC is allocated later for demand paging.

5.4.4.2 Analysis of MUMmer Results

A set of MUMmer experiments that determine the effect caused by various ESDC and main memory sizes is shown in Table 5.7. With abundant memory space, the execution times average just less than 2000 seconds. System time increases dramatically due to excessive paging as indicated by the increase in the number of major page faults. The correct functionality of ESDC in terms of caching pages backed by swap is shown in Table 5.7. As main memory capacity is reduced, there are increases in both the number of minor page faults and the number of read accesses through high memory bounce buffers (bounce reads). For all ESDC sizes lower than 256 MB, ESDC is entirely occupied by a large swap cache and pages backing files during most of the execution time. The MUMmer experiments are unique for causing large numbers of laundered pages, which are dirty pages that are cleaned by writing their contents to backing store. Lower main memory sizes cause millions of dirty pages to be laundered due to the high demand paging activity.

Some combinations of main memory and ESDC cause the execution time of MUMmer to increase dramatically. For example, with 128 MB of ESDC and 256 MB of main memory, the MUMmer application never finished because its resident pages were constantly being exchanged with backing store. This thrashing increases system response time, but the system was still usable. This behavior indicates that adequate amounts of both ESDC and main memory are required. The problem of the excessively long execution time of the above example could be solved either by adding ESDC or main memory.

Unlike in some other experiments, the performance of this experiment is affected more by main memory size than ESDC size. For example, the elapsed time increases moderately as main memory size approaches 256 MB. This indicates that, while many pages backed by swap are cached in ESDC, the application will likely replace these pages before they can effectively improve performance. While a large swap cache in ESDC would significantly improve performance of paging from backing store, no noticeable performance improvement was observed due to the memory access patterns of this application. These results indicate that a swap cache does not always improve demand paging performance. Instead, extended storage has the potential for further performance improvements if a portion of ESDC is reserved for paging in addition to its role as a page and swap cache.

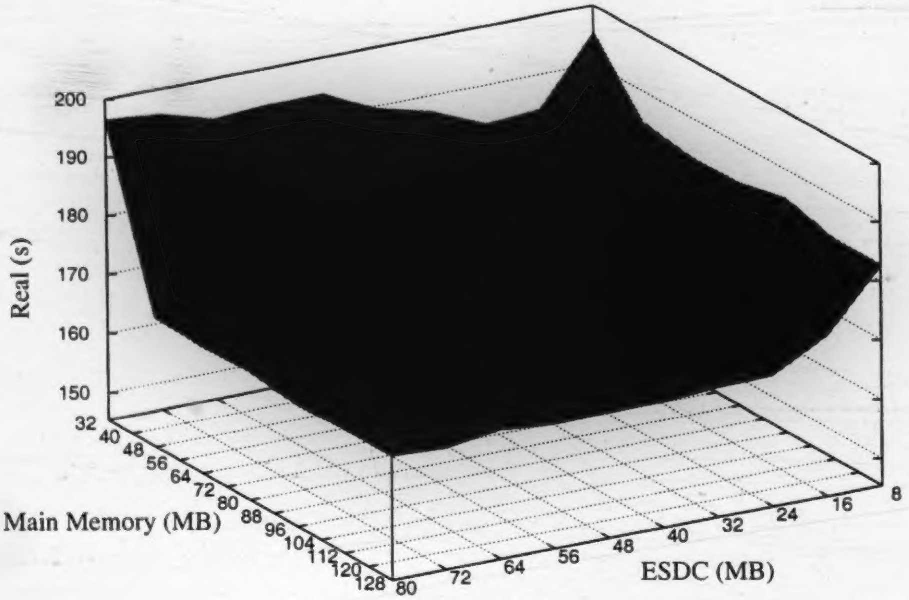


Figure 5.14: Execution time of Linux kernel compilation using cached ESDC.

5.4.5 Kernel Compilation Workload Results

ESDC should be evaluated with workloads that are similar to those produced by large office or engineering applications. Such programs are known for intensive virtual memory usage and frequent file system operations. Compilers are an excellent candidate for performing such operations because they frequently involve creating, reading, writing and deleting files [28]. A realistic application workload is the concurrent compilation of an unpatched Linux kernel (version 2.4.18) [70]. Creating multiple `gcc` compiler processes, a kernel build causes memory pressure for main memory sizes less than approximately 48 MB [10].

Each kernel compilation experiment requires two separate stages that are separated by a system reboot. The first stage involves creating a Linux kernel source tree on an empty file system. The kernel compilation is configured and all dependencies are generated as part of the initial stage. In the second stage, the execution time of the building of a kernel image is measured using the GNU `time` utility.

Table 5.8: Kernel Compilation Measurements for Uncached ESDC

ESDC Size (MB)	Access Time Ratio (%)	Real Time (s)	User Time (s)	System Time (s)
128	100	8252	8044	44
128	300	8261	8035	84
8	100	8220	8040	44
8	300	8266	8042	87

5.4.5.1 Analysis of Kernel Compilation Performance

The first kernel compilation experiment involved a system configuration that would help establish reference execution times. This experiment was run on a system using cached ESDC, with a file memory access time equivalent to that of conventional DRAM. The execution times are shown in Figure 5.14. The best performance is obtained for ESDC capacities larger than 40 MB and main memory sizes in excess of 56 MB. For this application, ESDC size is not the only consideration; low main memory sizes also have a negative effect on execution time.

To ensure accuracy of the access time penalties applied to the high memory zone used for ESDC, it is necessary to disable caching in this zone. Unfortunately, when a full kernel compilation is attempted for extended storage disk caches of various sizes with caching disabled, the execution time increases significantly. With caching enabled, the build completes in approximately three minutes. However, the execution time increases to over two hours when caching is disabled, regardless of the size of ESDC. One could speculate that compilation involves frequent accesses to certain files, such as header files, compiler and loader executables, and common makefiles. When lines belonging to the pages mapped to these files are cached in the SRAM memory of one of the processor caches, they tend to remain there due to the LRU cache replacement policies. Thus, compilation execution time decreases by an order of magnitude when processor caches are enabled.

Since kernel compilation relies heavily on the processor caches, this analysis can not include comparisons of performance for a range of ESDC and main memory

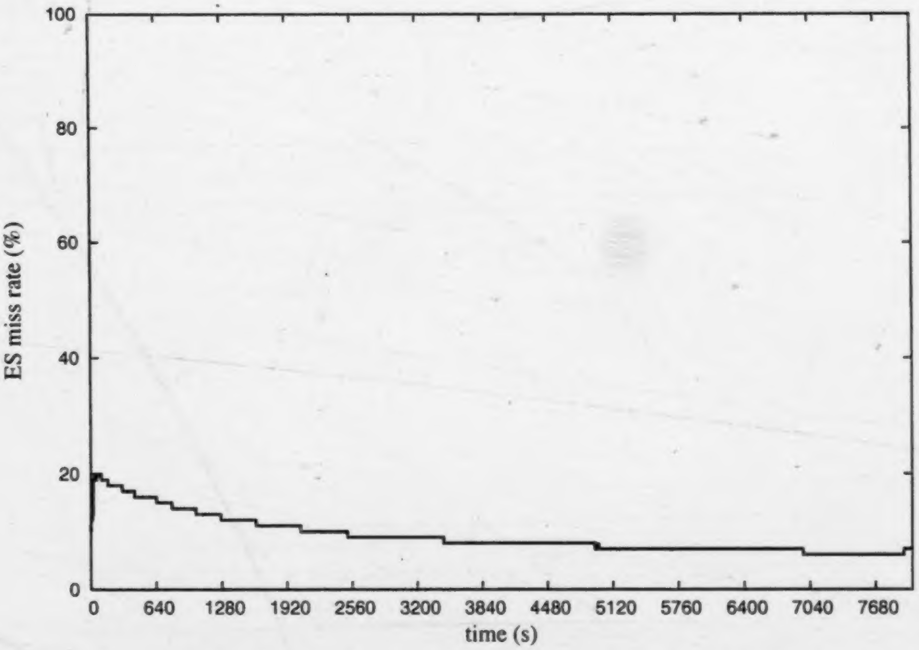


Figure 5.15: ESDC miss rate for kernel compilation with 128 MB of ESDC and 128 MB of main memory.

sizes. However, disabling caching still provides several interesting results for particular system memory configurations. Several different execution times are shown in Table 5.8, which are all obtained for experiments run while caching is disabled. As shown by these results, ESDC size has little effect on execution time. Disabling the processor caches significantly increases execution time, and this could obscure the benefit provided by increasing the amount of extended storage. However, since compilation is dominated by file system I/O, larger extended storage sizes have a beneficial effect, as shown in Figure 5.14.

Another interesting conclusion to make from the data present in Table 5.8 is the fact that tripling the file memory access time doubles the system time.[3] This conclusion makes sense because pages in extended storage are managed by the kernel and are never directly accessed by user processes. This fact is encouraging since system time is typically low relative to user time for most applications. The fact that user processes never directly access file memory ensures that algorithms in user space are not directly affected by the introduction of extended storage. Note that this relationship between file memory access time and system time could not be obtained without disabling caching.

---

[3] *System time* is the amount of processor time spent in kernel space.

5.4.5.2 Analysis of Kernel Compilation Miss Rates

An analysis of the ESDC miss rate will conclude this discussion of the Linux kernel compilation experiments. By polling the ESDC metrics via the `proc` file system every five seconds, the ESDC miss rate can be recorded throughout the compilation. The results are shown in Figure 5.15 for 128 MB of both ESDC and main memory and an access time ratio of 3. As the build starts, the miss rate increases because some of the most commonly used kernel source files are not yet cached in ESDC. After this, the miss rate decreases during the rest of the build, asymptotically approaching 7%. The miss rate for kernel compilation is at the opposite end of the spectrum when compared with the miss rate of the PostMark benchmark (see Section 5.4.2.4). The miss rate for the latter experiment with 128 MB of ESDC approaches 63%, indicating that the PostMark benchmark uses different the file system access patterns that are common to compilers—frequent and multiple accesses to the same set of files. The ESDC miss rate is very low for kernel compilation, emphasizing the importance of disk caches in modern desktop systems.

5.5 Conclusion

After verifying the implementation of ESDC, a suite of experiments was run to evaluate the performance of ESDC for different main memory and ESDC capacities as well as different file memory access time specifications. When ESDC uses file memory that is half the cost of conventional DRAM but suffers from an access time three times as long, the performance of PostMark can be improved by up to 36%. Equivalent performance measurements can be made when the working set size is equal to or larger than the size of ESDC. On the average, file memory with an access time ratio of 3 must be 37% and 31% larger than conventional DRAM to achieve equivalent performance for the PostMark and Bonnie benchmarks, respectively. The Bonnie benchmark revealed some issues with the quality of the official kernel implementation in terms of page cache metrics. Other than indicating minimum ESDC size requirements for reasonable execution times, MUMmer illustrated that not all applications benefit from ESDC demand paging support. Kernel compilation does not require large ESDC sizes for optimum performance but the low ESDC miss rate of 7% illustrates that disk caches are essential for optimum performance.

> The purpose of a disk cache is to improve performance; it has no other user visible function.
>
> — Alan J. Smith, 1985 [63]

Chapter 6

Conclusion

6.1 Introduction

The design and implementation of ESDC was based on several design principles that were established early during the research project. The design approach described in this thesis minimized the extent of the changes to the kernel to maintain stability and reduce the complexity caused by unknown side effects. One of the main disadvantages of the ESDC architecture is that it does not improve performance in all situations. Nevertheless, the results of a number of ESDC experiments show that it is possible to cost-effectively improve performance by using file memory as a general purpose extended storage disk cache.

6.2 Design Summary

6.2.1 Fundamentals

ESDC is designed under the assumption that, for file memory to be less expensive than conventional DRAM, it will have inferior overall performance. The performance disparity may be caused by an alternative memory technology such as multilevel memory. However, file memory could consist of portions of partially-good DRAM that function at the same speed as conventional DRAM. Areas of memories that contain faulty cells are recoverable at reduced performance, depending on the method of marking bad blocks. This thesis found that faulty blocks could be marked by an operating system's memory allocator with virtually no impact on performance if the blocks are multiples of a page frame in size.

Instead of relying on simulations, the memory hierarchy of an enterprise-class operating system was modified to evaluate file memory as extended storage. The scope of the investigation involved most of the memory management architecture, and several design decisions were possible due to this broad investigation. For example, while examining the virtual memory allocation algorithm, an innovative approach to file memory fault tolerance was devised that virtually eliminated the additional access time overhead associated with identifying bad blocks. Once the kernel memory management data structures were understood, ESDC design was straightforward since many of the elements of a new hierarchy stage were already present in the kernel. In addition, it was discovered that extended storage must be managed as a memory resource rather than as a physical storage device with mechanical limitations. This requires subtle changes to how the kernel allocates memory for I/O operations as well as for demand paging.

The extended storage disk cache was based on the presence of a dynamic virtual disk cache known as the page cache. By containing all pages belonging to the page cache in high memory, a contiguous area of memory can be reserved for ESDC. This is important for three reasons. First, when file memory is added to a system, it will almost certainly occupy a range of high physical memory addresses. Second, the containment of ESDC allowed for accurate file memory access time ratios by disabling the caching of file memory by the processor caches. Third, ESDC containment and caching control provides the ability to model large file memory banks that are accessible via an I/O or backplane bus instead of a memory bus. ESDC supports demand paging because the swap cache is a subset of the page cache. Since extended storage contains pages backed by files and pages associated with memory-mapped files, it is able to function as a true general purpose disk cache.

6.2.2 Limitations

There are a number of limitations with ESDC that could be addressed by future research. First, the current ESDC implementation is not portable. It was designed for the Linux operating system and depends on specific aspects of the Intel Pentium architecture. A related issue is the use of high memory for containing ESDC pages. The support for high memory in Linux is an interim and inefficient solution for the 4-GB addressing limitation of a 32-bit operating system. Linux eventually will evolve into a 64-bit operating system and potentially could drop high memory support. Re-introducing a high memory zone specifically for file memory is possible, but it would result in additional performance overhead. An alternative solution would be to investigate operating systems designed for non-uniform memory architectures (NUMA). While ESDC is contained in a fixed range of contiguous addresses, physical memory addresses may be interleaved among all memory modules for improved memory bandwidth in some systems. If this feature can not be disabled, it would complicate the straightforward adoption of file memory modules.

Another design limitation is the reduction in peak performance caused by converting the page cache to extended storage. This could be addressed by preserving the functionality of the page cache and adding a separate hierarchy stage. This approach, while requiring significantly more implementation effort, was used by Castro in his compressed virtual cache in 2003 [11]. In addition to the potential for peak performance improvements, a separate hierarchy stage would improve other aspects of ESDC design. For example, ESDC support of demand paging is limited to the pages added to the swap cache during page faulting. In situations such as swapping out processes during periods of high memory pressure, anonymous pages are added to the swap cache. These pages are not present in ESDC since they had not been originally allocated in high memory. Such issues could be addressed by adding a separate ESDC hierarchy stage below the existing virtual disk caches.

6.3 Summary of Results

6.3.1 PostMark Benchmark

The PostMark benchmark quantifies the cost-effectiveness of introducing file memory as an extended storage disk cache to a system that manages frequent transactions to many small files. Using legacy mainframes as a guideline, the file memory access time ratio for these experiments is 3, when normalized to conventional DRAM. To achieve equivalent I/O performance, an average of 37% more file memory than conventional DRAM must be purchased. This performance improvement is observed when the working set of the benchmark is larger than the ESDC size. Such a situation would be common on heavily loaded workstations or servers that process large numbers of small files.

Depending on the implementation, file memory could be substantially less expensive than DRAM. Assuming that this file memory is half the cost per bit of conventional DRAM, twice as much file memory can be purchased than the case where DRAM simply is added to the system. Introducing this quantity of file memory as ESDC can offer 36% higher throughput for read operations and 35% faster write operations without increasing costs. However, when excess ESDC capacity exists, the maximum achievable performance is 22% less than the peak performance possible if the file memory was replaced by an equivalent capacity of conventional DRAM. Enhancements to ESDC architecture could improve significantly on the current 22% reduction in peak performance.

Other PostMark results include the impact that file memory access times have on performance and ESDC miss rates. PostMark performance degrades linearly for larger file memory access time ratios, at a rate of up to 9.9% for each integer increase in the access time ratio. With small ESDC sizes, such as 16 MB, the ESDC miss rate when running PostMark approached 92%. For ESDC sizes above 96 MB, the ESDC miss rate never increased above 63%. This miss rate is not lower, likely due to the benchmark's emphasis on write I/O operations at the expense of reads. These results indicate that large ESDC sizes are essential for capturing most of the pages belonging to a working set.

6.3.2 Bonnie Benchmark

The Bonnie benchmark performs repeated block and character I/O operations on a single large file. Due to this behavior, it revealed two flaws present in the official implementation of the Linux 2.4.18 kernel. One of these bugs was repaired by removing an incomplete code sequence from the kernel without impacting stability or eliminating functionality. The other kernel bug caused race conditions to appear when large files are truncated during periods of intense memory pressure. With the exception of Bonnie, this causes errors of only a few percent in the experimental results.

The Bonnie and PostMark benchmark produced similar equivalent performance results when averaged over an 80-MB working set size. With a file memory access time ratio of 3, Bonnie requires an average of 31% more file memory than conventional DRAM for equivalent performance. Both benchmarks suggest that if file memory is at least 27% less expensive than DRAM, then ESDC can improve the performance of a system. However, when the Bonnie benchmark is executed with abundant file memory that is three times slower than DRAM, the peak random seek rate is lower by a factor of three than when file memory is replaced with DRAM. The sensitivity of ESDC to sustained raw I/O on single large files suggests that extended storage should supplement, rather than replace, virtual disk caches.

6.3.3 MUMmer Application

The MUMmer application requires large main memory capacities to align large genomes. This application performs few file I/O operations, so it was used to establish memory pressure and force demand paging. This application showed that most of the extended storage was used as a cache for pages in swap. The MUMmer experiments indicate that both ESDC size as well as a main memory size must be large enough to ensure that thrashing would not cause the execution time to increase exponentially. However, caching pages backed by swap during periods of intense memory pressure did not have an impact on ESDC execution time due to poor locality of reference of the memory access patterns of the MUMmer application.

To improve performance of applications such as MUMmer, the design of ESDC in terms of the swap cache requires several changes. First, pages backed by swap tend to be removed prematurely from the dynamic swap cache by the kernel, which is unnecessary for large extended storage capacities. Second, anonymous pages should also be cached in ESDC once they are backed by swap, which would reduce memory pressure in main memory and improve the performance of the MUMmer application. Finally, since ESDC currently is a cache for pages backed to disk, further performance improvements are possible if ESDC becomes a primary backing store for pages backed by swap.

6.3.4 Kernel Compilation Application

Concurrent compilation of the 2.4.18 Linux kernel is another application used to evaluate ESDC. Compilers make numerous accesses to the same file-backed pages, so disabling the processor caches is not feasible and file memory cost-effectiveness could not be evaluated using this application. Nevertheless, file memory access times have little effect on performance when high memory caching is disabled. The experiments show that a minimum ESDC size of 40 MB and at least 56 MB of main memory are necessary for the best compilation execution time. A normalized file memory access time of 3 will double the total system time of the application, which is reasonable since the slower file memory is accessed in kernel mode. The ESDC miss rate during kernel compilation are at most 7%, which is much lower than the the miss rate of the PostMark benchmark.

6.4 Conclusion

Edifices of computer architecture that were once confined to mainframes eventually appear in personal computer systems. Innovations such as multilevel caches, virtual memory and symmetric multiprocessing were used in mainframe systems before they became ubiquitous. Extended storage disk caches could have a similar future.

> The appealing aspect of disk cache is that it is a simple and elegant solution that avoids the need for many small and costly system changes and improvements.
>
> — Alan J. Smith, 1985 [63]

Bibliography

[1] S. Akyurek and K. Salem. Management of partially safe buffers. *IEEE Trans. on Computers*, 44(3):394–407, Mar 1995.

[2] D. Anderson. You don't know jack about disks. *ACM Queue*, 1(4), Jun 2003.

[3] Aries Electronics Data Sheet. *CSP-MicroBGA test and burn-in socket for devices from 14–27mm sq.*, 2003.

[4] D. H. Bailey. FFTs in external or hierarchical memory. In *Proc. of the ACM/IEEE Conference on Supercomputing*, pages 234–242, 1989.

[5] M. Baker et al. Non-volatile memory for fast, reliable file systems. In *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 10–22, Oct 1992.

[6] T. S. Barnett, M. Grady, K. Purdy, and A. D. Singh. Redundancy implications for early-life reliability: Experimental verification of an integrated yield-reliability model. In *Proc. of the International Test Conference*, pages 693–699, 2002.

[7] T. S. Barnett, A. D. Singh, and V. P. Nelson. Burn-in failures and local region yield: An integrated yield-reliability model. In *Proc. on the VLSI Test Symposium*, pages 326–332, May 2001.

[8] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly and Associates, 2nd edition, 2003.

[9] T. Bray. Bonnie. `http://www.textuality.com/bonnie/`, 1996.

[10] R. S. Castro. Adaptive compressed caching: Design and implementation. In *The 15th Symposium on Computer Architecture and High Performance Computing*, Nov 2003.

[11] R. S. Castro. Compressed caching. Master's thesis, University of Sao Paulo, 2003.

[12] C.-H. Chen and A. K. Somani. A cache protocol for error detection and recovery in fault-tolerant computing systems. In *Proc. of the Twenty-Fourth International Symposium on Fault-Tolerant Systems*, pages 278–287, Jun 1994.

[13] P. M. Chen, C. M. Aycock, W. T. Ng, G. Rajamani, and R. Sivaramakrishnan. Rio: Storing files reliably in memory. Technical Report CSE-TR250-95, University of Michigan, Jul 1995.

[14] P. M. Chen, W. T. Ng, G. Rajamani, and C. M. Aycock. The Rio file cache: Surviving operating systems crashes. In *Proc. of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 74–83, Oct 1996.

[15] E. I. Cohen, G. M. King, and J. T. Brady. Storage hierarchies. *IBM Systems Journal*, 28(1):62–76, 1989.

[16] A. Davidson. MEMS-actuated magnetic probe-based storage. In *Digest of the Asia-Pacific Magnetic Recording Conference*, pages CE3–01–CE3–02, Aug 2002.

[17] D. Deese. Expanded storage management with MVS/ESA. In *Proc. of the Computer Measurement Group*, pages 780–792, 1993.

[18] A. Delcher, A. Phillippy, S. Salzberg, and S. Kurtz. MUMmer. The Institute for Genomic Research, `http://www.tigr.org/software/mummer/`, 2003.

[19] A. L. Delcher, A. Phillippy, J. Carlton, and Steven Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.

[20] E. D. Demaine and J. I. Munro. Fast allocation and deallocation with an improved buddy system. In *Foundations of Software Technology and Theoretical Computer Science*, pages 84–96, 1999.

[21] E. Fujiwara and M. Tanaka. A defect-tolerant WSI file memory system using address permutation scheme for spare allocation. In *Proc. of the IEEE Inter-*

national Workshop on Fault Tolerance in VLSI Systems, pages 183–190, Oct 1993.

[22] S. Galvin. *Operating System Concepts*. Addison Wesley, 4th edition, 1994.

[23] P. K. Garvin. Method and system for managing bad areas in flash memory. United States Patent 6260156, Jul 2001.

[24] M. Gorman. Understanding the Linux memory manager. Master's thesis, University of Limerick, 2003.

[25] J. Hennessy and D. Patterson. *Computer architecture: A quantitative approach*. Morgan Kaufmann, 3rd edition, 2003.

[26] C. Ho. All about ARAM (Audio RAM). `http://www.simmtester.com/page/news/showpubnews.asp?title=All+About+ARA%M+(Audio+Ram)&num=5`, May 1994.

[27] C. Ho. Innovative testing puts fallout DRAM back into systems, CST Inc. `http://www.simmtester.com/page/news/showpubnews.asp?title=All+About+ARA%M+(Audio+Ram)&num=5`, Jan 2003.

[28] Y. Hu and Q. Yang. DCD—Disk caching disk: A new approach for boosting I/O. In *Proc. of the Twenty-Third International Symposium on Computer Architecture*, pages 169–178, May 1996.

[29] Y. Hu and Q. Yang. A new hierarchical disk architecture. *IEEE Micro*, 18:64–76, 1998.

[30] S. H. Hwang and G. S. Choi. On-chip cache memory resilience. In *Proc. of the Third IEEE International High-Assurance Systems Engineering Symposium*, pages 240–247, Nov 1998.

[31] Silicon Graphics Inc. Cray research incorporated. `http://www.new-npac.org/projects/html/projects/cdroms/cewes-1999-06-vol%1/nhse/hpccsurvey/orgs/crayri/crayri.html`.

[32] Intel Corporation. *Intel Architecture Software Developer's Manual: Volume 3: System Programming*, 1999.

[33] C. Joly. Yield improvement in semiconductor memory through ternary content

addressable memories. Master's thesis, University of Alberta, 2003.

[34] H. L. Kalter et al. A 50-ns 16-Mb DRAM with a 10-ns data rate and on-chip ECC. *IEEE J. of Solid-State Circuits*, 25(5):1118–1127, Oct 1990.

[35] S. F. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation.* PhD thesis, University of Texas at Austin, 1999.

[36] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *IEEE Computer*, 27(3):38–46, Mar 1994.

[37] H. H. Kari, H. K. Saikkonen, N. Park, and F. Lombardi. Analysis of repair algorithms for mirrored-disk systems. *IEEE Trans. on Reliability*, 46(2):193–200, Jun 1997.

[38] K. Katayama et al. File memory device and information processing apparatus using the same. United States Patent 6351787, Feb 2002.

[39] J. Katcher. PostMark: A new filesystem benchmark. Technical Report TR3022, Network Appliance, Inc., Oct 1997.

[40] M. Y. Kim and A. N. Tantawi. Asynchronous disk interleaving: Approximating access delays. *IEEE Trans. on Computers*, 40(7):801–810, Jul 1991.

[41] K. C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, Oct 1965.

[42] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997.

[43] J. Kohl, M. Stonebraker, and C. Staelin. Highlight: A file system for tertiary storage. In *Proc. of the Twelfth IEEE Symposium on Mass Storage Systems*, pages 157–161, 1993.

[44] K. Lee et al. A 1-Gbit synchronous dynamic random access memory with an independent subarray-controlled scheme and a hierarchical decoding scheme. *IEEE J. of Solid-States Circuits*, 33(5):779–786, May 1998.

[45] K. Li and K. Petersen. Evaluation of memory system extensions. In *Proc. of the 18th International Symposium on Computer Architecture*, pages 84–93, May 1991.

[46] C. Morganti and T. Chen. Graceful capacity degradation for ultra-large hierarchical memory structures. In *Proc. of the IFIP International Conference on Hardware Description Languages*, pages 817–822, Aug 1995.

[47] W. T. Ng and P. M. Chen. The design and verification of the Rio file cache. *IEEE Trans. on Computers*, 50(4):322–337, Apr 2001.

[48] T. Nightingale, Y. Hu, and Q. Yang. The design and implementation of a DCD device driver for Unix. In *Proc. of the USENIX Annual Technical Conf.*, 1999.

[49] V. Oklobdzija, editor. *The Computer Engineering Handbook*, chapter 80. CRC Press, 2000.

[50] N. Prasad and J. Savit. *IBM mainframes: Architecture and design*. McGraw-Hill, 2nd edition, 1994.

[51] B. Prince. *Semiconductor Memories: A Handbook of Design, Manufacture, and Application*. John Wiley, 2nd edition, 1991.

[52] S. Quinlan. A cached WORM file system. In *Software - Practice and Experience*, pages 1289–1299, 1991.

[53] E. Rahm. Performance evaluation of extended storage architectures for transaction processing. In *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 308–317, Jun 1992.

[54] Reactive Computer Services Data Sheet. *K3M FFD 2.5" IDE Plus*, 2003.

[55] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proc. of the USENIX Annual Technical Conf.*, 2000.

[56] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. on Computer Systems*, 10(1):26–52, 1992.

[57] A. Rubini. *Linux Device Drivers*. O'Reilly and Associates, 2nd edition, 2001.

[58] C. Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, Mar 1994.

[59] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation

of a log-structured file system for UNIX. In *Proc. of the USENIX Annual Technical Conf.*, pages 307–326, Jan 1993.

[60] P. P. Shirvani and E. J. McCluskey. PADded cache: A new fault-tolerance technique for cache memories. In *Proc. of VLSI Test Symposium*, pages 440–445, Apr 1999.

[61] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proc. of the 21st International Conf. on Computer Design (ICCD)*, Oct 2003.

[62] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, Sep 1982.

[63] A. J. Smith. Disk cache–Miss ratio analysis and design considerations. *ACM Trans. on Computer Systems*, 3(3):161–203, Aug 1985.

[64] A. K. Somani and K. S. Trivedi. A cache error propagation model. In *Proc. of the Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 15–21, Dec 1997.

[65] C. H. Stapper. Synergistic fault-tolerance for memory chips. *IEEE Trans. on Computers*, 41(9):1078–1087, Sep 1992.

[66] C. H. Stapper, J. A. Fifield, H. L. Kalter, and W. A. Klaasen. High-reliability fault-tolerant 16-Mbit memory chip. *IEEE Trans. on Reliability*, 42(4):596–603, Dec 1993.

[67] C. H. Stapper, A. N. McLaren, and M. Dreckmann. Yield model for productivity optimization of VLSI memory chips with redundancy and partially good product. *IBM J. Res. Develop.*, 24(3):398–409, May 1980.

[68] K. Sugawara, K. Nakamura, M. Matoba, and S. Sakairi. Semiconductor file memory. United States Patent 4958323, Sep 1990.

[69] A. Tal. *Two technologies compared: NOR vs. NAND White Paper*. M-Systems, Jul 2003.

[70] L. Torvalds. The Linux kernel archives. `http://www.kernel.org`, Feb 2002.

[71] Toshiba Corp. *TC58V100 1-Gbit CMOS NAND E2PROM Datasheet*, Mar 2001.

[72] Toshiba Corp. *TC58V128 128-Mbit CMOS NAND E2PROM Datasheet*, Mar 2001.

[73] Toshiba Corp. *TC58V256 256-Mbit CMOS NAND E2PROM Datasheet*, May 2001.

[74] Toshiba Corp. *TC58V512 512-Mbit CMOS NAND E2PROM Datasheet*, Mar 2001.

[75] Toshiba Corp. *TC58V64 64-Mbit CMOS NAND E2PROM Datasheet*, Oct 2001.

[76] Toshiba Corp. *What is NAND flash memory?*, Mar 2003.

[77] S. G. Tucker. The IBM 3090 system: An overview. *IBM Systems Journal*, 25(1):4–19, 1986.

[78] S. A. Ung. Design and evaluation of a variable-capacity multilevel DRAM test chip. Master's thesis, University of Alberta, 2004.

[79] A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proc. of the USENIX Annual Technical Conf.*, pages 15–28, Jun 2002.

[80] Western Digital. *Western Digital EIDE Hard Drives*, 2001.

[81] C. Wickman. File store memories. Master's thesis, University of Alberta, 2000.

[82] C. Wickman, D. G. Elliott, and B. F. Cockburn. Cost models for large file memory DRAMs with ECC and bad block marking. In *International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 319–327, Nov 1999.

[83] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proc. of the USENIX Annual Technical Conf.*, pages 101–116, 1999.

[84] S. J. E. Wilton and N. P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE J. of Solid-States Circuits*, 31(5):677–688, May 1996.

[85] Y. Xiang, B. F. Cockburn, and D. G. Elliott. Design of a multilevel DRAM with adjustable cell capacity. *Canadian J. of Electrical and Computer Engineering*, 26(2):55–59, Apr 2001.

[86] J. Young. *Exploring IBM's New Age Mainframes.* Maximum Press, 1996.

Appendix A

Implementation

A.1 Selected Kernel Modifications for ESDC

This section provides supplementary material to the discussion of the design of ESDC. It is not intended to serve as a complete technical report of ESDC design. It offers detailed explanations of some of the technical issues behind several important design decisions. First, changes in the kernel are required to configure arbitrary high memory capacities. Second, the mechanisms for controlling the allocation of high memory pages are needed to constrain ESDC pages to high memory. Third, the method of controlling file memory access time involves new functions that reduce the performance of page transfers between low and high memory. Finally, ESDC access metrics are obtain in the same way that the kernel computes other metrics.

A.1.1 High Memory Emulation

ESDC may require physical high memory or high memory emulation, depending on the requested sizes of the different memory zones. Therefore, it was necessary to modify the use of `MAXMEM` preprocessor directive to adjust the size of an emulated high memory zone. The kernel defines `MAXMEM` as 896 MB, which is the result of Equation (4.1). This directive was replaced so that the total main memory used during high memory emulation could be specified by a kernel boot parameter. The minimum permissible setting is 32 MB to provide sufficient memory for the first two zones while the maximum setting is 896 MB. The original definition of

MAXMEM, shown in Listing A.1, was not modified since the value is used for verifying that at most 896 MB of main memory is specified. To ensure that enough physical memory was actually available to meet the requested amount of memory, the BIOS was used to calculate the amount of physical memory. The changes ESDC required for configuring high memory emulation are shown in listing A.2.

Listing A.1: include/asm-i386/page.h - MAXMEM definition

```
/*
 * ...
 *
 * A __PAGE_OFFSET of 0xC0000000 means that the kernel has
 * a virtual address space of one gigabyte, which limits the
 * amount of physical memory you can use to about 950MB.
 *
 * If you want more physical memory than this then see the CONFIG_HIGHMEM4G
 * and CONFIG_HIGHMEM64G options in the kernel configuration.
 */

#define __PAGE_OFFSET           (0xC0000000)

/*
 * This much address space is reserved for vmalloc() and iomap()
 * as well as fixmap mappings.
 */
#define _VMALLOC_RESERVE        (128 << 20)

...

#define MAXMEM ((unsigned long)(-PAGE_OFFSET-VMALLOC_RESERVE))
```

Listing A.2: arch/i386/kernel/setup.c - High memory setup for ESDC

```
        /* esdc: returns parsed memory size */
        esdc_mem_size = parse_mem_cmdline(cmdline_p);

        /* esdc: support for new 'esdc=' kernel parameter */
        esdc_highmem_size = parse_esdc_cmdline(cmdline_p);

#define PFN_UP(x)       (((x) + PAGE_SIZE-1) >> PAGE_SHIFT)
#define PFN_DOWN(x)     ((x) >> PAGE_SHIFT)
#define PFN_PHYS(x)     ((x) << PAGE_SHIFT)

/*
 * Reserved space for vmalloc and iomap - defined in asm/page.h
 */
/* #define MAXMEM_PFN   PFN_DOWN(MAXMEM) */  /* esdc: removed */
#define MAX_NONPAE_PFN  (1 << 20)

        /*
         * partially used pages are not usable - thus
         * we are rounding upwards:
         */
        start_pfn = PFN_UP(__pa(&_end));

        /*
         * Find the highest page frame number we have available
         */
        max_pfn = 0;
```

```
for (i = 0; i < e820.nr_map; i++) {
        unsigned long start, end;
        /* RAM? */
        if (e820.map[i].type != E820_RAM)
                continue;
        start = PFN_UP(e820.map[i].addr);
        end = PFN_DOWN(e820.map[i].addr + e820.map[i].size);
        if (start >= end)
                continue;
        if (end > max_pfn)
                max_pfn = end;
}

/* if no mem= kernel parameter supplied, use max memory available, or */
/* validate mem= kernel parameter to make sure it is not too large */
if (esdc_mem_size == 0)
{
        printk("esdc: using BIOS to calculate total physical RAM\n");
        esdc_mask = ~( (1<<20) - 1);   /* round up to nearest MB */
        esdc_mem_size = (PFN_PHYS(max_pfn) + ~esdc_mask) & esdc_mask;
}

if (esdc_highmem_size > esdc_mem_size)
{
        printk("esdc: bad ESDC size of %lld bytes\n",
                esdc_highmem_size);
        esdc_highmem_size = 32 * (1 << 20);
}

esdc_maxmem = esdc_mem_size - esdc_highmem_size;

/* use original def'n of MAXMEM for sanity check of
 * esdc_maxmem */
if (esdc_maxmem > MAXMEM)
{
        esdc_maxmem = MAXMEM;
        esdc_highmem_size = esdc_mem_size - esdc_maxmem;
        printk("esdc: kernel only addresses %lld bytes\n",
                esdc_maxmem);
        printk("esdc: ESDC size adjusted\n");
}
printk("esdc: total visible RAM:  %lld bytes\n", esdc_mem_size);
printk("esdc: validated ESDC size: %lld bytes\n", esdc_highmem_size);

esdc_maxmem_pfn = PFN_DOWN(esdc_maxmem);  /* esdc: not a define now */
```

A.1.2 Utilizing GFP Flags for ESDC

The investigation of the get free page (GFP) flags for high memory allocation and the changes required to support ESDC is discussed in this section. The primary GFP flag that determines whether or not a page will reside in high memory is the `__GFP_HIGHMEM` zone modifier. While this flag is used explicitly in various cases, implicit use is more common by means of the `GFP_HIGHUSER` bitmask. The only difference between the `GFP_HIGHUSER` and the more common `GFP_USER` bitmasks is the presence of the `__GFP_HIGHMEM` flag.

Table A.1: `__GFP_HIGHMEM` Analysis

Kernel Source File	Use of `__GFP_HIGHMEM` symbol	Action
`fs/ntfs/support.h`	__vmalloc() function call where second parameter is (GFP_NOFS | __GFP_HIGHMEM)	Remove
`mm/page_alloc.c`	__GFP_HIGHMEM used in memory allocation zone lists	Retain
`include/linux/mm.h`	#define __GFP_HIGHMEM 0x02	Retain
`include/linux/mm.h`	#define GFP_HIGHUSER (__GFP_WAIT | __GFP_IO | __GFP_HIGHIO | __GFP_FS | __GFP_HIGHMEM)	Retain
`include/linux/vmalloc.h`	__vmalloc() function call where second parameter is (GFP_KERNEL | __GFP_HIGHMEM)	Remove
`arch/mips/mm/umap.c`	__vmalloc() function call where second parameter is (GFP_KERNEL | __GFP_HIGHMEM)	Retain
`arch/mips64/mm/umap.c`	__vmalloc() function call where second parameter is (GFP_KERNEL | __GFP_HIGHMEM)	Retain

An exhaustive search of the kernel sources for use of the `__GFP_HIGHMEM` symbol gave the results shown in Table A.1. The flag was removed from the bitmask in the `support.h` and `vmalloc.h` files since `vmalloc()` is responsible for allocating memory from the virtual address space. This memory is mapped to kernel space as a contiguous range of virtual addresses and is not visible from user space. Therefore, these instances of the `__GFP_HIGHMEM` flag must be removed to help ensure that the high memory zone is only used for page cache allocations. The instance of the flag in `page_alloc.c` is necessary since it occurs in the portion of the buddy system of memory allocation that supports memory zoning. The function involved, `build_zonelists()`, creates several lists of memory zones where the first element in each list is the preferred zone and subsequent elements are fallback zones. The directives in `mm.h` are simply the definitions of the flag itself and the `GFP_HIGHUSER` bitmask. Finally, the instances of the flag in `umap.c` are not relevant since the file is intended for the MIPS architecture.

ESDC pages are backed by files and should be allocated in the high memory zone. If a page is backed by a file, it is associated with an inode. An inode object contains an `address_space` object for block device files. One of the functions of an `address_space` object is to identify a page in the page cache. This object includes an integer field called `gfp_mask` that can be set to one of the the memory allocation flags for the owner of the pages. During initialization of the

Table A.2: `__GFP_HIGHUSER` Analysis

Kernel Function	Use of `__GFP_HIGHUSER` bitmask	Action
do_wp_page()	Bitmask is set for a new page when the old page is copied to the new page during a Copy On Write	Change
do_anonymous_page()	Bitmask is set for a new anonymous page. An anonymous page is not mapped to a file file so such a page does not belong in ESDC memory.	Change
do_no_page()	Bitmask is set during an early COW break.	Change
copy_strings()	Bitmask is set when creating pages for `execvp()` data. Changing this instance removed the last few percent user pages from ESDC memory.	Change
clean_inode()	The inode field `gfp_mask` is set to the `GFP_HIGHUSER` bitmask upon initialization of a new inode. This is required for ESDC page containment. In addition, `page_cache_alloc()` is called with an inode mapping and the `gfp_mask` parameter is passed to `alloc_pages()`	Retain
read_swap_cache_async()	Bitmask is set for a new page created when reading from swap.	Retain

`address_space` object for a new inode (see Table A.2), this field is set to the `GFP_HIGHUSER` bitmask. Therefore, this instance of the `GFP_HIGHUSER` bitmask ensures that ESDC pages are contained in the high memory zone. Other occurrences shown in Table A.2 were changed to prevent user process pages from appearing in high memory.

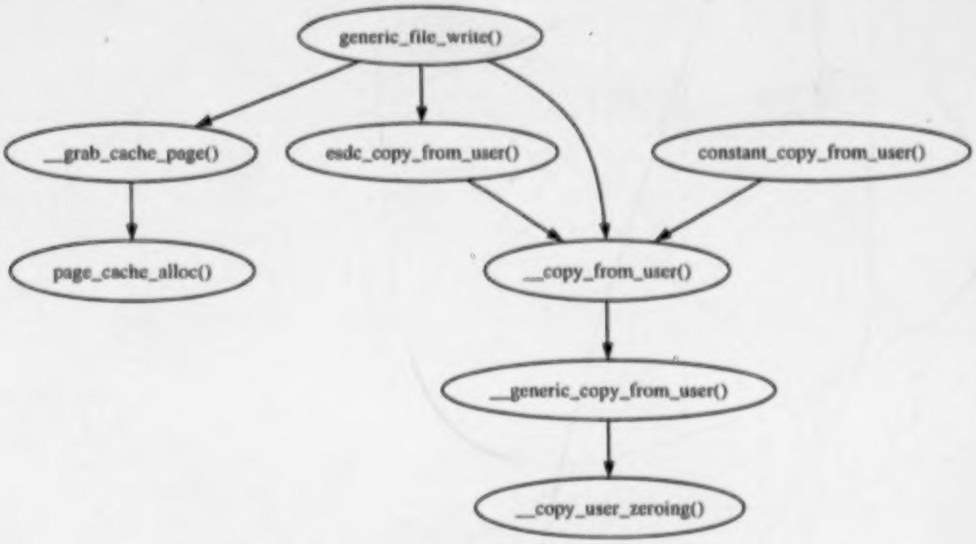

generic_file_write()
__grab_cache_page()
esdc_copy_from_user()
constant_copy_from_user()
page_cache_alloc()
__copy_from_user()
__generic_copy_from_user()
__copy_user_zeroing()

Figure A.1: Function callgraph of `generic_file_write()`

A.1.3 Configurable Performance Implementation

To determine the functions that are involved with copying pages from user space, it was necessary to investigate the functions involved with `__copy_from_user`. It was found that the function `generic_file_write()` copies pages from user space to ESDC memory in kernel space by calling `__copy_from_user()`, as shown in Figure A.1. It is important to note that, since all block writes to disk use the page cache, all block writes pass through ESDC. The kernel modification to introduce configurable ESDC performance penalties involved inserting a function (`esdc_copy_from_user`) in this callgraph as shown in Figure A.1.

A general read operation involves copying pages from ESDC in kernel space to processes in user space.[1] The function `file_read_actor()` calls the function `__copy_to_user()`, as shown in Figure A.2. This callgraph was modified for ESDC so that the former function would call `esdc_copy_to_user()`, which would use the specified performance penalty in its calls to the `__copy_to_user` function. Not only do these functions permit configurable performance, but they

---

[1] Note that if the page was not originally in ESDC, it would be read from disk and then added to ESDC before the copy occurred.

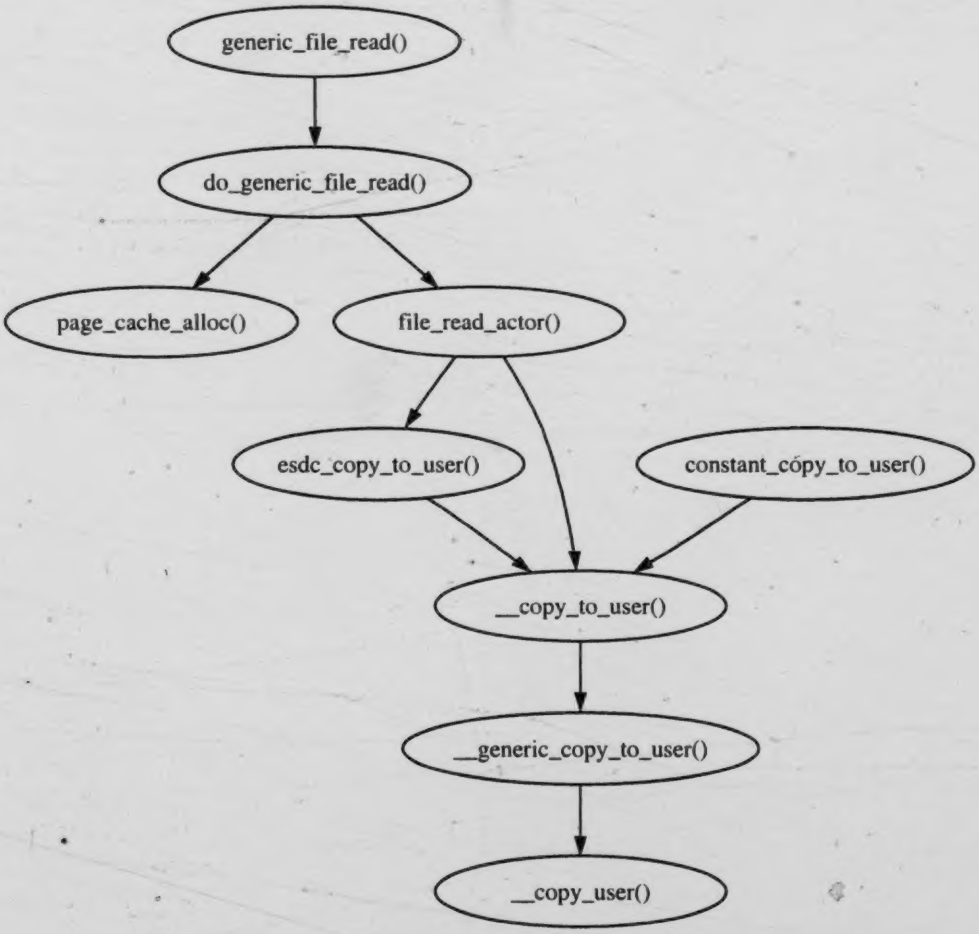

generic_file_read()
do_generic_file_read()
page_cache_alloc()
file_read_actor()
esdc_copy_to_user()
constant_copy_to_user()
__copy_to_user()
__generic_copy_to_user()
__copy_user()

Figure A.2: Function callgraph of `generic_file_read()`

update the ESDC metrics used in the calculation of ESDC hit and miss rates.

The implementation of the configurable performance penalties used repeated copies of the page to artificially increase the access time of file memory. As discussed in Chapter 4, this requires disabling caching in high memory for accurate access time ratios. The implementation of configurable performance for writing to ESDC is shown the function `esdc_copy_from_user()` in Listing A.3. If the access time ratio of file memory normalized to DRAM is 1, the penalty is expressed as 100% since it is difficult to specify integers mixed with floating point in the `proc` file system. With an access time ratio of 1, the page is copied from user space to high memory a single time. For larger access time ratios, the number of copies is equal to the access time ratio, rounded to the nearest integer. For fractional access time ratios, a portion of a page cannot be copied, so the fractional value is estimated by repeated copies from the page to a temporary array variable. Since this is not optimized assembly that normally is used to copy pages, the estimation will be somewhat pessimistic. The same method is used for configurable performance for reading from ESDC, which is handled by the `esdc_copy_to_user()` function shown in Listing A.4.

Listing A.3: mm/filemap.c - Configurable performance for ESDC writes

```
long esdc_copy_from_user(char *addr, char *buf, unsigned size)
{
        long result = 0;
        int i;
        int integer_penalty;
        float fractional_penalty;
        int bucket[10];

        /* convert percentage to delay factor */
        integer_penalty = esdc_prm.esdc_un.es_from_mem / 100;
        fractional_penalty = (esdc_prm.esdc_un.es_from_mem % 100) / 100.0;

        if (fractional_penalty)
            for (i = 0; i < (unsigned long)(fractional_penalty * size); i++)
                bucket[i%10] = (int) *(buf+i);

        for (i = 0; i < integer_penalty; i++)
                result = __copy_from_user(addr, buf, size);

        atomic_inc(&esdc_nr_writes);
        esdc_copy_diag('<');

        return result;
}
```

Listing A.4: mm/filemap.c - Configurable performance for ESDC reads

```
unsigned long esdc_copy_to_user(char *buf, char *addr, unsigned
long size)
{
        unsigned long result = 0;
        int i;
        int integer_penalty;
        float fractional_penalty;
        int bucket[10];

        /* convert percentage to delay factor */
        integer_penalty = esdc_prm.esdc_un.es_to_mem / 100;
        fractional_penalty = (esdc_prm.esdc_un.es_to_mem % 100) /
100.0;

        if (fractional_penalty)
            for (i = 0; i < (unsigned long)(fractional_penalty *
size); i++)
                bucket[i%10] = (int) *(addr+i);

        for (i = 0; i < integer_penalty; i++)
```

```
                result = __copy_to_user(buf, addr, size);

        atomic_inc(&esdc_nr_reads);
        esdc_copy_diag('>');

        return result;
}
```

A.1.4 ESDC Access Statistics

The page cache hash table is necessary for the kernel to efficiently determine if a page is present in ESDC. It first appears that a large number of kernel functions are involved with searching the page cache hash table. However, an investigation of the function call tree revealed that all of these functions eventually call a single function: `__find_page_nolock()` in Listing A.5. This function quickly searches the page cache hash table to determine if the supplied page is in the page cache. The atomic counters for cumulative hits and misses on this hash table are updated in this function. It is important to distinguish the hit and miss rates for the hash table from those for the actual page cache.

Listing A.5: mm/filemap.c - Updating ESDC access statistics

```
static inline struct page * __find_page_nolock(struct
address_space *mapping, unsigned long offset, struct page *page)
{
        goto inside;

        for (;;) {
                page = page->next_hash;
inside:
                if (!page)
                {
                        /* atomically increment ESDC miss count */
                        atomic_inc(&esdc_nr_hash_misses);
                        goto not_found;
                }
                if (page->mapping != mapping)
                        continue;
                if (page->index == offset)
                {
                        /* atomically increment ESDC hit count */
                        atomic_inc(&esdc_nr_hash_hits);
                        break;
                }
        }

not_found:
        return page;
}
```

Appendix B

Operating System Modifications

B.1 ESDC Kernel Patch

Listing B.1: ESDC patch for the 2.4.18 Linux kernel

```
diff -ruN linux/arch/i386/kernel/setup.c stage/arch/i386/kernel/setup.c
--- linux/arch/i386/kernel/setup.c   2002-02-25 12:37:53.000000000 -0700
+++ stage/arch/i386/kernel/setup.c   2004-01-26 01:05:03.059991000 -0700
@@ -705,11 +705,12 @@
 } /* setup_memory_region */


-static void __init parse_mem_cmdline (char ** cmdline_p)
+static unsigned long long __init parse_mem_cmdline (char ** cmdline_p)
 {
    char c = ' ', *to = command_line, *from = COMMAND_LINE;
    int len = 0;
    int usermem = 0;
+   unsigned long long mem_size = 0;

    /* Save unparsed command line copy for /proc/cmdline */
    memcpy(saved_command_line, COMMAND_LINE, COMMAND_LINE_SIZE);
@@ -738,7 +739,8 @@
                 * blow away any automatically generated
                 * size
                 */
-               unsigned long long start_at, mem_size;
+               unsigned long long start_at;
+               printk("esdc: using kernel parameter 'mem='\n");

                if (usermem == 0) {
                    /* first time in: zap the whitelist
@@ -777,12 +779,54 @@
        printk(KERN_INFO "user-defined physical RAM map:\n");
        print_memory_map("user");
    }
+
+   if (mem_size)
+       return mem_size + HIGH_MEMORY;  /* esdc: requested size in bytes */
+   else
+       return 0;
+}
+
+static unsigned long long __init parse_esdc_cmdline (char ** cmdline_p)
+{
+   char c = ' ', *to = command_line, *from = COMMAND_LINE;
+    int len = 0;
+    unsigned long long esdc_size = 32 * (1 << 20);
+
+    for (;;) {
+        /*
+         * "esdc=XXX[kKmM]" defines a high memory size for an ESDC
+         */
+        if (c == ' ' && !memcmp(from, "esdc=", 5)) {
+            if (to != command_line)
+                to--;
+            /* The ESDC size is equivalent to the
+             * size of the high memory zone.
+             */
+            esdc_size = memparse(from+5, &from);
+            printk("esdc: using kernel parameter 'esdc='\n");
+        }
+        c = *(from++);
+        if (!c)
+            break;
+        if (COMMAND_LINE_SIZE <= ++len)
+            break;
+        *(to++) = c;
+    }
+    *to = '\0';
+    *cmdline_p = command_line;
+
+    return esdc_size;
 }

 void __init setup_arch(char **cmdline_p)
 {
     unsigned long bootmap_size, low_mem_size;
-    unsigned long start_pfn, max_pfn, max_low_pfn;
+    unsigned long start_pfn, max_pfn, max_low_pfn = 0;
+    unsigned long esdc_maxmem_pfn = 0;       /* esdc: was MAXMEM_PFN */
+    unsigned long long esdc_maxmem;      /* esdc: was MAXMEM */
+    unsigned long long esdc_mem_size;    /* esdc: size of accessable mem */
+    unsigned long long esdc_highmem_size; /* esdc: size of high mem zone*/
+    unsigned long long esdc_mask;        /* esdc: mask for fixing mem size*/
     int i;

 #ifdef CONFIG_VISWS
@@ -820,7 +864,11 @@
     data_resource.start = virt_to_bus(&_etext);
     data_resource.end = virt_to_bus(&_edata)-1;

-    parse_mem_cmdline(cmdline_p);
+    /* esdc: returns parsed memory size */
+    esdc_mem_size = parse_mem_cmdline(cmdline_p);
+
+    /* esdc: support for new 'esdc=' kernel parameter */
+    esdc_highmem_size = parse_esdc_cmdline(cmdline_p);

 #define PFN_UP(x)    (((x) + PAGE_SIZE-1) >> PAGE_SHIFT)
 #define PFN_DOWN(x)     ((x) >> PAGE_SHIFT)
@@ -829,7 +877,7 @@
 /*
  * Reserved space for vmalloc and iomap - defined in asm/page.h
  */
-#define MAXMEM_PFN PFN_DOWN(MAXMEM)
+/* #define MAXMEM_PFN  PFN_DOWN(MAXMEM) */  /* esdc: removed */
 #define MAX_NONPAE_PFN (1 << 20)

     /*
@@ -855,16 +903,50 @@
                 max_pfn = end;
 	}
 
+	/* if no mem= kernel parameter supplied, use max memory available, or */
+	/* validate mem= kernel parameter to make sure it is not too large */
+	if (esdc_mem_size == 0)
+	{
+		printk("esdc: using BIOS to calculate total physical RAM\n");
+		esdc_mask = ~((1<<20) - 1);	/* round up to nearest MB */
+		esdc_mem_size = (PFN_PHYS(max_pfn) + ~esdc_mask) & esdc_mask;
+	}
+
+	if (esdc_highmem_size > esdc_mem_size)
+	{
+		printk("esdc: bad ESDC size of %lld bytes\n",
+			esdc_highmem_size);
+		esdc_highmem_size = 32 * (1 << 20);
+	}
+
+	esdc_maxmem = esdc_mem_size - esdc_highmem_size;
+
+	/* use original def'n of MAXMEM for sanity check of esdc_maxmem */
+	if (esdc_maxmem > MAXMEM)
+	{
+		esdc_maxmem = MAXMEM;
+		esdc_highmem_size = esdc_mem_size - esdc_maxmem;
+		printk("esdc: kernel only addresses %lld bytes\n",
+			esdc_maxmem);
+		printk("esdc: ESDC size adjusted\n");
+	}
+	printk("esdc: total visible RAM:   %lld bytes\n", esdc_mem_size);
+	printk("esdc: validated ESDC size: %lld bytes\n",
+		esdc_highmem_size);
+
+	esdc_maxmem_pfn = PFN_DOWN(esdc_maxmem);  /* esdc: not a define now */
+
+
 	/*
 	 * Determine low and high memory ranges:
 	 */
 	max_low_pfn = max_pfn;
-	if (max_low_pfn > MAXMEM_PFN) {
-		max_low_pfn = MAXMEM_PFN;
+	if (max_low_pfn > esdc_maxmem_pfn) {
+		max_low_pfn = esdc_maxmem_pfn;
 #ifndef CONFIG_HIGHMEM
 		/* Maximum memory usable is what is directly addressable */
 		printk(KERN_WARNING "Warning only %ldMB will be used.\n",
-					MAXMEM>>20);
+					esdc_maxmem>>20);
 		if (max_pfn > MAX_NONPAE_PFN)
 			printk(KERN_WARNING "Use a PAE enabled kernel.\n");
 		else
@@ -882,8 +964,8 @@
 
 #ifdef CONFIG_HIGHMEM
 	highstart_pfn = highend_pfn = max_pfn;
-	if (max_pfn > MAXMEM_PFN) {
-		highstart_pfn = MAXMEM_PFN;
+	if (max_pfn > esdc_maxmem_pfn) {
+		highstart_pfn = esdc_maxmem_pfn;
 		printk(KERN_NOTICE "%ldMB HIGHMEM available.\n",
 			pages_to_mb(highend_pfn - highstart_pfn));
 	}
diff -ruN linux/arch/i386/mm/init.c stage/arch/i386/mm/init.c
--- linux/arch/i386/mm/init.c	2001-12-21 10:41:53.000000000 -0700
+++ stage/arch/i386/mm/init.c	2003-08-05 14:31:56.319999000 -0600
```

5

```
@@ -598,6 +598,27 @@
     return;
 }

+void si_esdc_info(struct esdc_info *val)
+{
+    val->esdc_io_requests = atomic_read(&esdc_nr_io_requests);
+    val->esdc_bounce_reads = atomic_read(&esdc_nr_bounce_reads);
+    val->esdc_bounce_writes = atomic_read(&esdc_nr_bounce_writes);
+    val->esdc_reads = atomic_read(&esdc_nr_reads);
+    val->esdc_writes = atomic_read(&esdc_nr_writes);
+    val->esdc_hash_hits = atomic_read(&esdc_nr_hash_hits);
+    val->esdc_hash_misses = atomic_read(&esdc_nr_hash_misses);
+    val->esdc_dirty = atomic_read(&esdc_nr_dirty);
+    val->esdc_anonymous = atomic_read(&esdc_nr_anonymous);
+    val->esdc_still_cached = atomic_read(&esdc_nr_still_cached);
+    val->esdc_drop_buffers = atomic_read(&esdc_nr_drop_buffers);
+    val->esdc_will_swap = atomic_read(&esdc_nr_will_swap);
+    val->esdc_remove_cache = atomic_read(&esdc_nr_remove_cache);
+    val->esdc_truncd_cache = atomic_read(&esdc_nr_truncd_cache);
+    val->esdc_freed_pages = atomic_read(&esdc_nr_freed_pages);
+    val->esdc_allocd_pages = atomic_read(&esdc_nr_allocd_pages);
+    return;
+}
+
 #if defined(CONFIG_X86_PAE)
 struct kmem_cache_s *pae_pgd_cachep;
 void __init pgtable_cache_init(void)
diff -ruN linux/drivers/block/ll_rw_blk.c stage/drivers/block/ll_rw_blk.c
--- linux/drivers/block/ll_rw_blk.c 2002-02-25 12:37:57.000000000 -0700
+++ stage/drivers/block/ll_rw_blk.c 2003-08-03 12:50:46.009996000 -0600
@@ -123,6 +123,11 @@
  */
 static int queue_nr_requests, batch_requests;

+/*
+ * esdc: esdc number of io requests
+ */
+atomic_t esdc_nr_io_requests = ATOMIC_INIT(0);
+
 static inline int get_max_sectors(kdev_t dev)
 {
     if (!max_sectors[MAJOR(dev)])
@@ -668,6 +673,15 @@
     bh = create_bounce(rw, bh);
 #endif

+    /* esdc: monitor how many pages associated with the buffers
+     * actually are in page cache */
+    if (bh->b_page->mapping)
+        if (esdc_prm.esdc_un.io_monitor > 0)
+        atomic_inc(&esdc_nr_io_requests);
+    /*      printk("+");    */   /* in page cache */
+    /* else
+        printk("-"); */
+
 /* look for a free request. */
    /*
     * Try to coalesce the new request with old requests
diff -ruN linux/fs/exec.c stage/fs/exec.c
--- linux/fs/exec.c 2001-12-21 10:41:55.000000000 -0700
+++ stage/fs/exec.c 2003-08-13 14:19:27.759994000 -0600
@@ -207,7 +207,7 @@
                page = bprm->page[i];
                new = 0;
                if (!page) {
```

```
-               page = alloc_page(GFP_HIGHUSER);
+               page = alloc_page(GFP_USER);  /* esdc: was GFP_HIGHUSER */
                bprm->page[i] = page;
                if (!page)
                        return -ENOMEM;
diff -ruN linux/fs/inode.c stage/fs/inode.c
--- linux/fs/inode.c      2001-12-21 10:41:55.000000000 -0700
+++ stage/fs/inode.c      2004-01-16 23:31:19.159999000 -0700
@@ -785,6 +785,8 @@
     inode->i_cdev = NULL;
     inode->i_data.a_ops = &empty_aops;
     inode->i_data.host = inode;
+    /* esdc: keep GFP_HIGHUSER for page cache in high memory
+     * essc: change to GPF_USER to contain swap cache in high memory */
     inode->i_data.gfp_mask = GFP_HIGHUSER;
     inode->i_mapping = &inode->i_data;
 }
diff -ruN linux/fs/ntfs/support.h stage/fs/ntfs/support.h
--- linux/fs/ntfs/support.h 2002-02-25 12:38:09.000000000 -0700
+++ stage/fs/ntfs/support.h 2003-04-01 13:44:39.000000000 -0700
@@ -53,7 +53,8 @@
         BUG();
     }
     if (size >> PAGE_SHIFT < num_physpages)
-        return __vmalloc(size, GFP_NOFS | __GFP_HIGHMEM, PAGE_KERNEL);
+        /* esdc: __GFP_HIGHMEM removed */
+        return __vmalloc(size, GFP_NOFS, PAGE_KERNEL);
     return NULL;
 }

diff -ruN linux/fs/proc/proc_misc.c stage/fs/proc/proc_misc.c
--- linux/fs/proc/proc_misc.c     2001-11-20 22:29:09.000000000 -0700
+++ stage/fs/proc/proc_misc.c     2003-12-20 16:53:41.389985000 -0700
@@ -156,35 +156,40 @@
      * The above will go away eventually, once the tools
      * have been updated.
      */
+    /* esdc: this entire sprintf was modified to output stats
+     *       in units of pages as well as KB */
     len += sprintf(page+len,
-        "MemTotal:     %8lu kB\n"
-        "MemFree:      %8lu kB\n"
-        "MemShared:    %8lu kB\n"
-        "Buffers:      %8lu kB\n"
-        "Cached:       %8lu kB\n"
-        "SwapCached:   %8lu kB\n"
-        "Active:       %8u kB\n"
-        "Inactive:     %8u kB\n"
-        "HighTotal:    %8lu kB\n"
-        "HighFree:     %8lu kB\n"
-        "LowTotal:     %8lu kB\n"
-        "LowFree:      %8lu kB\n"
-        "SwapTotal:    %8lu kB\n"
-        "SwapFree:     %8lu kB\n",
-        K(i.totalram),
-        K(i.freeram),
-        K(i.sharedram),
-        K(i.bufferram),
-        K(pg_size - swapper_space.nrpages),
-        K(swapper_space.nrpages),
-        K(nr_active_pages),
-        K(nr_inactive_pages),
-        K(i.totalhigh),
-        K(i.freehigh),
-        K(i.totalram-i.totalhigh),
-        K(i.freeram-i.freehigh),
-        K(i.totalswap),
-        K(i.freeswap));
+        "MemTotal:     %8lu kB        %8lu pages\n"
+        "MemFree:      %8lu kB        %8lu pages\n"
+        "MemShared:    %8lu kB        %8lu pages\n"
+        "Buffers:      %8lu kB        %8lu pages\n"
+        "Cached:       %8lu kB        %8lu pages\n"
+        "SwapCached:   %8lu kB        %8lu pages\n"
+        "Active:       %8u kB        %8u pages\n"
+        "Inactive:     %8u kB        %8u pages\n"
+        "HighTotal:    %8lu kB        %8lu pages\n"
+        "HighFree:     %8lu kB        %8lu pages\n"
+        "Overflow:     %8ld kB        %8ld pages\n"   /* esdc */
+        "LowTotal:     %8lu kB        %8lu pages\n"
+        "LowFree:      %8lu kB        %8lu pages\n"
+        "SwapTotal:    %8lu kB        %8lu pages\n"
+        "SwapFree:     %8lu kB        %8lu pages\n",
+        K(i.totalram), i.totalram,
+        K(i.freeram), i.freeram,
+        K(i.sharedram), i.sharedram,
+        K(i.bufferram), i.bufferram,
+        K(pg_size - swapper_space.nrpages), pg_size - swapper_space.nrpages,
+        K(swapper_space.nrpages), swapper_space.nrpages,
+        K(nr_active_pages), nr_active_pages,
+        K(nr_inactive_pages), nr_inactive_pages,
+        K(i.totalhigh), i.totalhigh,
+        K(i.freehigh), i.freehigh,
+        K((pg_size - swapper_space.nrpages) - (i.totalhigh - i.freehigh)),
/*esdc*/
+        (pg_size - swapper_space.nrpages) - (i.totalhigh - i.freehigh), /*esdc*/
+        K(i.totalram-i.totalhigh), i.totalram-i.totalhigh,
+        K(i.freeram-i.freehigh), i.freeram-i.freehigh,
+        K(i.totalswap), i.totalswap,
+        K(i.freeswap), i.freeswap);

    return proc_calc_metrics(page, start, off, count, eof, len);
 #undef B
@@ -410,6 +415,94 @@
    return proc_calc_metrics(page, start, off, count, eof, len);
 }

+
+static int esdc_read_proc(char *page, char **start, off_t off,
+                int count, int *eof, void *data)
+{
+   struct esdc_info i;
+   int len;
+   double hit_rate;
+   double miss_rate;
+
+   memset((char *)&i, 0, sizeof(struct esdc_info));
+
+   si_esdc_info(&i);
+
+   /* avoid meaningless hits metric */
+   if ((i.esdc_writes < i.esdc_bounce_writes) ||
+       (i.esdc_reads < i.esdc_bounce_reads))
+       i.esdc_hits = 0;
+   else
+       i.esdc_hits = (i.esdc_writes - i.esdc_bounce_writes) +
+           (i.esdc_reads - i.esdc_bounce_reads);
+
+   /* misses are more straighforward */
+   i.esdc_misses = i.esdc_bounce_writes + i.esdc_bounce_reads;
+
+   /* find the hit rate and miss rates */
+    hit_rate = (((double)i.esdc_hits)
+        / (double)(i.esdc_writes+i.esdc_reads)) * 100;
+    miss_rate = (double) ( (((double)i.esdc_misses)
+        / (double)(i.esdc_writes+i.esdc_reads)) * 100);
+
+    /* now make copies and truncate */
+    i.esdc_hit_rate = (unsigned long) hit_rate;
+    i.esdc_miss_rate = (unsigned long) miss_rate;
+
+    /* add one if we need to round up */
+    if ((hit_rate - (double) i.esdc_hit_rate) >= 0.5)
+        i.esdc_hit_rate++;
+    if ((miss_rate - (double) i.esdc_miss_rate) >= 0.5)
+        i.esdc_miss_rate++;
+
+    /* avoid meaningless percentages */
+    if (i.esdc_hit_rate > 100) i.esdc_hit_rate = 100;
+    if (i.esdc_miss_rate > 100) i.esdc_miss_rate = 100;
+
+    len = sprintf(page, "IO requests:                       %8lu\n"
+                "IO bounce reads:                 %8lu\n"
+                "IO bounce writes:                %8lu\n"
+                "ES reads:                        %8lu\n"
+                "ES writes:                       %8lu\n"
+                "ES hits:                         %8lu\n"
+                "ES misses:                       %8lu\n"
+                "ES hit rate:                     %8lu\n"
+                "ES miss rate:                    %8lu\n"
+                "ES hash table hits:              %8lu\n"
+                "ES hash table misses:            %8lu\n"
+                "Num dirty writepages():          %8lu\n"
+                "Num anonymous pages w/ buffers:  %8lu\n"
+                "Num pages still in page cache:   %8lu\n"
+                "Num pages w/ buffers not freed:  %8lu\n"
+                "Num swap_outs for anon pages:    %8lu\n"
+                "Num pages del from page cache:   %8lu\n"
+                "Num pages truncd page cache:     %8lu\n"
+                "Num pages freed from highmem:    %8lu\n"
+                "Num pages alloc'd in highmem:    %8lu\n",
+                i.esdc_io_requests,
+                i.esdc_bounce_reads,
+                i.esdc_bounce_writes,
+                i.esdc_reads,
+                i.esdc_writes,
+                i.esdc_hits,
+                i.esdc_misses,
+                i.esdc_hit_rate,
+                i.esdc_miss_rate,
+                i.esdc_hash_hits,
+                i.esdc_hash_misses,
+                i.esdc_dirty,
+                i.esdc_anonymous,
+                i.esdc_still_cached,
+                i.esdc_drop_buffers,
+                i.esdc_will_swap,
+                i.esdc_remove_cache,
+                i.esdc_truncd_cache,
+                i.esdc_freed_pages,
+                i.esdc_allocd_pages);
+
+    return proc_calc_metrics(page, start, off, count, eof, len);
+}
+
 /*
  * This function accesses profiling information. The returned data is
  * binary: the sampling step and the actual contents of the profile
```

```
@@ -526,6 +619,7 @@
        {"swaps",    swaps_read_proc},
        {"iomem",    memory_read_proc},
        {"execdomains", execdomains_read_proc},
+       {"esdc",     esdc_read_proc},
        {NULL,}
    };
    for (p = simple_ones; p->name; p++)
diff -ruN linux/include/asm-i386/page.h stage/include/asm-i386/page.h
--- linux/include/asm-i386/page.h    2002-02-25 12:38:12.000000000 -0700
+++ stage/include/asm-i386/page.h    2003-04-02 10:55:48.000000000 -0700
@@ -125,7 +125,7 @@

 #define PAGE_OFFSET         ((unsigned long)__PAGE_OFFSET)
 #define VMALLOC_RESERVE         ((unsigned long)__VMALLOC_RESERVE)
-#define __MAXMEM        (-__PAGE_OFFSET-__VMALLOC_RESERVE)
+#define __MAXMEM        (-__PAGE_OFFSET-__VMALLOC_RESERVE) /* esdc:XXX */
 #define MAXMEM          ((unsigned long)(-PAGE_OFFSET-VMALLOC_RESERVE))
 #define __pa(x)             ((unsigned long)(x)-PAGE_OFFSET)
 #define __va(x)             ((void *)((unsigned long)(x)+PAGE_OFFSET))
diff -ruN linux/include/linux/kernel.h stage/include/linux/kernel.h
--- linux/include/linux/kernel.h     2002-02-25 12:38:13.000000000 -0700
+++ stage/include/linux/kernel.h     2003-08-05 21:26:47.120005000 -0600
@@ -179,6 +179,51 @@
    unsigned long freehigh;     /* Available high memory size */
    unsigned int mem_unit;      /* Memory unit size in bytes */
    char _f[20-2*sizeof(long)-sizeof(int)]; /* Padding: libc5 uses this.. */
+
+};
+
+/* fields are number of events or number of pages as appropriate */
+struct esdc_info {
+   /* # io requests */
+   unsigned long esdc_io_requests;
+   /* # copies to highmem from bounce buffers */
+   unsigned long esdc_bounce_reads;
+   /* # copies from highmem to bounce buffers */
+   unsigned long esdc_bounce_writes;
+   /* # es read using copy to user */
+   unsigned long esdc_reads;
+   /* # es write using copy from user */
+   unsigned long esdc_writes;
+   /* # es hits (calculated) */
+   unsigned long esdc_hits;
+   /* # es misses (calculated) */
+   unsigned long esdc_misses;
+   /* # es hit rate (calculated) */
+   unsigned long esdc_hit_rate;
+   /* # es miss rate (calculated) */
+   unsigned long esdc_miss_rate;
+   /* # es page cache hash table hits */
+   unsigned long esdc_hash_hits;
+   /* # es page cache hash table misses */
+   unsigned long esdc_hash_misses;
+   /* # writepage calls for dirty mapped page */
+   unsigned long esdc_dirty;
+   /* # anonymous pages with buffers */
+   unsigned long esdc_anonymous;
+   /* # pages still in page cache (bufs freed) */
+   unsigned long esdc_still_cached;
+   /* # pages w/ buffers that can't be released */
+   unsigned long esdc_drop_buffers;
+   /* # swap_out() calls for anon process pages */
+   unsigned long esdc_will_swap;
+   /* # pages removed from page cache (shrink) */
+   unsigned long esdc_remove_cache;
+       /* # pages removed from page cache (truncd) */
+       unsigned long esdc_truncd_cache;
+       /* # pages freed from highmem (buddy sys) */
+       unsigned long esdc_freed_pages;
+       /* # pages allocated in highmem (buddy sys) */
+       unsigned long esdc_allocd_pages;
 };

 #endif
diff -ruN linux/include/linux/mm.h stage/include/linux/mm.h
--- linux/include/linux/mm.h     2001-12-21 10:42:03.000000000 -0700
+++ stage/include/linux/mm.h     2004-01-26 01:05:03.139992000 -0700
@@ -447,6 +447,7 @@
 extern void mem_init(void);
 extern void show_mem(void);
 extern void si_meminfo(struct sysinfo * val);
+extern void si_esdc_info(struct esdc_info * val);
 extern void swapin_readahead(swp_entry_t);

 extern struct address_space swapper_space;
@@ -515,6 +516,49 @@
 extern unsigned long page_unuse(struct page *);
 extern void truncate_inode_pages(struct address_space *, loff_t);

+extern atomic_t esdc_nr_io_requests;
+extern atomic_t esdc_nr_bounce_reads;
+extern atomic_t esdc_nr_bounce_writes;
+extern atomic_t esdc_nr_reads;
+extern atomic_t esdc_nr_writes;
+extern atomic_t esdc_nr_hash_hits;
+extern atomic_t esdc_nr_hash_misses;
+extern atomic_t esdc_nr_dirty;
+extern atomic_t esdc_nr_anonymous;
+extern atomic_t esdc_nr_still_cached;
+extern atomic_t esdc_nr_drop_buffers;
+extern atomic_t esdc_nr_will_swap;
+extern atomic_t esdc_nr_remove_cache;
+extern atomic_t esdc_nr_truncd_cache;
+extern atomic_t esdc_nr_freed_pages;
+extern atomic_t esdc_nr_allocd_pages;
+
+/* the parameter block for esdc. If you add or
+ * remove any of the parameters, make sure to update kernel/sysctl.c
+ * and the documentation at linux/Documentation/sysctl/vm.txt.
+ */
+
+#define ESDC_N_PARAM 9
+
+union esdc_param {
+    struct {
+         int es_to_mem;       /* % delay factor copy to user */
+         int es_from_mem;     /* % delay factor copy from user*/
+         int es_to_dev;       /* % delay factor copy to dev */
+         int es_from_dev;     /* % delay factor copy from dev */
+         int diag_str_len;    /* diagnostic print str length */
+         int io_monitor;      /* monitor ll io requests */
+         int dummy3;      /* unused */
+         int dummy4;      /* unused */
+         int dummy5;      /* unused */
+    } esdc_un;
+    unsigned int data[ESDC_N_PARAM];
+};
+
+extern union esdc_param esdc_prm;
+extern int esdc_min[];
+extern int esdc_max[];
```

```
+
 /* generic vm_area_ops exported for stackable file systems */
 extern int filemap_sync(struct vm_area_struct *, unsigned long,    size_t,
unsigned int);
 extern struct page *filemap_nopage(struct vm_area_struct *, unsigned long,
int);
diff -ruN linux/include/linux/swap.h stage/include/linux/swap.h
--- linux/include/linux/swap.h  2001-11-22 12:46:19.000000000 -0700
+++ stage/include/linux/swap.h  2003-07-14 15:16:44.799997000 -0600
@@ -92,6 +92,8 @@
 extern atomic_t buffermem_pages;
 extern spinlock_t pagecache_lock;
 extern void __remove_inode_page(struct page *);
+extern void esdc_overflow_info(zone_t *zone, int balance_classzone,
+   int verbose);

 /* Incomplete types for prototype declarations: */
 struct task_struct;
diff -ruN linux/include/linux/sysctl.h stage/include/linux/sysctl.h
--- linux/include/linux/sysctl.h     2001-11-26 06:29:17.000000000 -0700
+++ stage/include/linux/sysctl.h     2003-04-07 12:39:38.000000000 -0600
@@ -141,7 +141,8 @@
     VM_PGT_CACHE=9,      /* struct: Set page table cache parameters */
     VM_PAGE_CLUSTER=10, /* int: set number of pages to swap together */
         VM_MIN_READAHEAD=12,     /* Min file readahead */
-        VM_MAX_READAHEAD=13      /* Max file readahead */
+        VM_MAX_READAHEAD=13,      /* Max file readahead */
+        VM_ESDC=14        /* esdc configuration */
 };


 diff -ruN linux/include/linux/vmalloc.h stage/include/linux/vmalloc.h
 --- linux/include/linux/vmalloc.h     2001-11-22 12:46:20.000000000 -0700
 +++ stage/include/linux/vmalloc.h     2003-04-07 12:39:38.000000000 -0600
 @@ -32,7 +32,8 @@

  static inline void * vmalloc (unsigned long size)
  {
 -   return __vmalloc(size, GFP_KERNEL | _GFP_HIGHMEM, PAGE_KERNEL);
 +   /* return __vmalloc(size, GFP_KERNEL | _GFP_HIGHMEM, PAGE_KERNEL); */
 +   return __vmalloc(size, GFP_KERNEL, PAGE_KERNEL);  /* esdc */
  }

  /*
 diff -ruN linux/kernel/sysctl.c stage/kernel/sysctl.c
 --- linux/kernel/sysctl.c     2001-12-21 10:42:04.000000000 -0700
 +++ stage/kernel/sysctl.c     2003-02-01 18:09:52.000000000 -0700
 @@ -275,6 +275,9 @@
     &vm_min_readahead, sizeof(int), 0644, NULL, &proc_dointvec},
     {VM_MAX_READAHEAD, "max-readahead",
     &vm_max_readahead, sizeof(int), 0644, NULL, &proc_dointvec},
 +   {VM_ESDC, "esdc_ctl", &esdc_prm, 9*sizeof(int), 0644, NULL,
 +    &proc_dointvec_minmax, &sysctl_intvec, NULL,
 +    &esdc_min, &esdc_max},
     {0}
  };

 diff -ruN linux/mm/filemap.c stage/mm/filemap.c
 --- linux/mm/filemap.c   2002-02-25 12:38:13.000000000 -0700
 +++ stage/mm/filemap.c   2003-12-20 16:53:41.299988000 -0700
 @@ -44,6 +44,11 @@
    */

  atomic_t page_cache_size = ATOMIC_INIT(0);
 +atomic_t esdc_nr_hash_hits = ATOMIC_INIT(0);
 +atomic_t esdc_nr_hash_misses = ATOMIC_INIT(0);
```

```
+atomic_t esdc_nr_writes = ATOMIC_INIT(0);
+atomic_t esdc_nr_reads = ATOMIC_INIT(0);
+atomic_t esdc_nr_truncd_cache = ATOMIC_INIT(0);
 unsigned int page_hash_bits;
 struct page **page_hash_table;

@@ -52,6 +57,12 @@
 EXPORT_SYMBOL(vm_max_readahead);
 EXPORT_SYMBOL(vm_min_readahead);

+union esdc_param esdc_prm = {{100, 100, 100, 100, 0, 1, 0, 0, 0}};
+
+/* The min and max parameter values that we will allow to be assigned */
+int esdc_min[ESDC_N_PARAM]={   50,    50,    50,    50,     0,     0,    0,  0,  0};
+int esdc_max[ESDC_N_PARAM]={64000,64000,64000,64000,    80,   100,    0,  0,  0};
+

 spinlock_t pagecache_lock __cacheline_aligned_in_smp = SPIN_LOCK_UNLOCKED;
 /*
@@ -203,6 +214,7 @@

        __lru_cache_del(page);
        __remove_inode_page(page);
+       /* atomic_inc(&esdc_nr_truncd_cache);   */
        UnlockPage(page);
        page_cache_release(page);
        continue;
@@ -247,6 +259,7 @@
    ClearPageDirty(page);
    ClearPageUptodate(page);
    remove_inode_page(page);
+   atomic_inc(&esdc_nr_truncd_cache);
    page_cache_release(page);
 }

@@ -443,11 +456,19 @@
        page = page->next_hash;
 inside:
        if (!page)
+       {
+           /* atomically increment ESDC miss count */
+           atomic_inc(&esdc_nr_hash_misses);
            goto not_found;
+       }
        if (page->mapping != mapping)
            continue;
        if (page->index == offset)
+       {
+           /* atomically increment ESDC hit count */
+           atomic_inc(&esdc_nr_hash_hits);
            break;
+       }
    }

 not_found:
@@ -1544,6 +1565,87 @@
    return retval;
 }

+/* esdc_copy_diag
+ *
+ * Diagnose calls to esdc_copy_????_user
+ *
+ */
+static inline void esdc_copy_diag(char c)
+{
```

```
+    static int i = 0;
+
+    if (esdc_prm.esdc_un.diag_str_len == 0)
+        return;
+    printk("%c", c);
+    if (i < esdc_prm.esdc_un.diag_str_len - 1)
+        i++;
+    else
+    {
+        printk("\n");
+        i = 0;
+    }
+}
+
+/* esdc_copy_to_user
+ *
+ * Delay element by repeating calls to __copy_to_user
+ *
+ */
+unsigned long esdc_copy_to_user(char *buf, char *addr, unsigned long size)
+{
+    unsigned long result = 0;
+    int i;
+    int integer_penalty;
+    float fractional_penalty;
+    int bucket[10];
+
+    /* convert percentage to delay factor */
+    integer_penalty = esdc_prm.esdc_un.es_to_mem / 100;
+    fractional_penalty = (esdc_prm.esdc_un.es_to_mem % 100) / 100.0;
+
+    if (fractional_penalty)
+        for (i = 0; i < (unsigned long)(fractional_penalty * size); i++)
+        bucket[i%10] = (int) *(addr+i);
+
+    for (i = 0; i < integer_penalty; i++)
+        result = __copy_to_user(buf, addr, size);
+
+    atomic_inc(&esdc_nr_reads);
+    esdc_copy_diag('>');
+
+    return result;
+}
+
+/* esdc_copy_from_user
+ *
+ * Delay element by repeating calls to __copy_from_user
+ *
+ */
+long esdc_copy_from_user(char *addr, char *buf, unsigned size)
+{
+    long result = 0;
+    int i;
+    int integer_penalty;
+    float fractional_penalty;
+    int bucket[10];
+
+    /* convert percentage to delay factor */
+    integer_penalty = esdc_prm.esdc_un.es_from_mem / 100;
+    fractional_penalty = (esdc_prm.esdc_un.es_from_mem % 100) / 100.0;
+
+    if (fractional_penalty)
+        for (i = 0; i < (unsigned long)(fractional_penalty * size); i++)
+        bucket[i%10] = (int) *(buf+i);
+
+    for (i = 0; i < integer_penalty; i++)
+       result = __copy_from_user(addr, buf, size);
+
+   atomic_inc(&esdc_nr_writes);
+   esdc_copy_diag('<');
+
+   return result;
+}
+
 int file_read_actor(read_descriptor_t * desc, struct page *page, unsigned long
offset, unsigned long size)
 {
    char *kaddr;
@@ -1553,7 +1655,7 @@
        size = count;

    kaddr = kmap(page);
-   left = __copy_to_user(desc->buf, kaddr + offset, size);
+   left = esdc_copy_to_user(desc->buf, kaddr + offset, size);
    kunmap(page);

    if (left) {
@@ -3001,7 +3103,7 @@
        status = mapping->a_ops->prepare_write(file, page, offset, offset+bytes);
        if (status)
            goto sync_failure;
-       page_fault = __copy_from_user(kaddr+offset, buf, bytes);
+       page_fault = esdc_copy_from_user(kaddr+offset, (char *)buf, bytes);
        flush_dcache_page(page);
        status = mapping->a_ops->commit_write(file, page, offset, offset+bytes);
        if (page_fault)
diff -ruN linux/mm/highmem.c stage/mm/highmem.c
--- linux/mm/highmem.c  2001-12-21 10:42:05.000000000 -0700
+++ stage/mm/highmem.c  2003-12-12 14:19:13.249997000 -0700
@@ -37,6 +37,8 @@
 pte_t * pkmap_page_table;

 static DECLARE_WAIT_QUEUE_HEAD(pkmap_map_wait);
+atomic_t esdc_nr_bounce_reads = ATOMIC_INIT(0);
+atomic_t esdc_nr_bounce_writes = ATOMIC_INIT(0);

 static void flush_all_zero_pkmaps(void)
 {
@@ -209,12 +211,30 @@
 {
    struct page *p_from;
    char *vfrom;
+   int i;
+   int integer_penalty;
+   int fractional_penalty;
+   int bucket[10];
+
+   /* convert perceantage to delay factor */
+   integer_penalty = esdc_prm.esdc_un.es_to_dev / 100;
+   fractional_penalty = (esdc_prm.esdc_un.es_to_dev % 100) / 100.0;

    p_from = from->b_page;

    vfrom = kmap_atomic(p_from, KM_USER0);
-   memcpy(to->b_data, vfrom + bh_offset(from), to->b_size);
+
+   if (fractional_penalty)
+       for (i = 0; i < (unsigned long)(fractional_penalty * to->b_size); i++)
+       bucket[i%10] = (int) *(vfrom + bh_offset(from) + i);
+
+   for (i = 0; i < integer_penalty; i++)
+       memcpy(to->b_data, vfrom + bh_offset(from), to->b_size);
+
 	kunmap_atomic(vfrom, KM_USER0);
+
+	if (esdc_prm.esdc_un.io_monitor > 0)
+		atomic_inc(&esdc_nr_bounce_writes);
 }

 static inline void copy_to_high_bh_irq (struct buffer_head *to,
@@ -223,14 +243,32 @@
 	struct page *p_to;
 	char *vto;
 	unsigned long flags;
+	int i;
+	int integer_penalty;
+	int fractional_penalty;
+	int bucket[10];
+
+	/* convert perceantage to delay factor */
+	integer_penalty = esdc_prm.esdc_un.es_from_dev / 100;
+	fractional_penalty = (esdc_prm.esdc_un.es_from_dev % 100) / 100.0;

 	p_to = to->b_page;
 	__save_flags(flags);
 	__cli();
 	vto = kmap_atomic(p_to, KM_BOUNCE_READ);
-	memcpy(vto + bh_offset(to), from->b_data, to->b_size);
+
+	if (fractional_penalty)
+		for (i = 0; i < (unsigned long)(fractional_penalty * to->b_size); i++)
+		bucket[i%10] = (int) *(from->b_data + i);
+
+	for (i = 0; i < integer_penalty; i++)
+		memcpy(vto + bh_offset(to), from->b_data, to->b_size);
+
 	kunmap_atomic(vto, KM_BOUNCE_READ);
 	__restore_flags(flags);
+
+	if (esdc_prm.esdc_un.io_monitor > 0)
+		atomic_inc(&esdc_nr_bounce_reads);
 }

 static inline void bounce_end_io (struct buffer_head *bh, int uptodate)
diff -ruN linux/mm/memory.c stage/mm/memory.c
--- linux/mm/memory.c	2002-02-25 12:38:13.000000000 -0700
+++ stage/mm/memory.c	2003-02-25 17:37:07.000000000 -0700
@@ -965,7 +965,7 @@
 	page_cache_get(old_page);
 	spin_unlock(&mm->page_table_lock);

-	new_page = alloc_page(GFP_HIGHUSER);
+	new_page = alloc_page(GFP_USER);  /* esdc: was GFP_HIGHUSER */
 	if (!new_page)
 		goto no_mem;
 	copy_cow_page(old_page,new_page,address);
@@ -1195,7 +1195,7 @@
 		/* Allocate our own private page. */
 		spin_unlock(&mm->page_table_lock);

-		page = alloc_page(GFP_HIGHUSER);
+		page = alloc_page(GFP_USER);  /* esdc: was GFP_HIGHUSER */
 		if (!page)
 			goto no_mem;
 		clear_user_highpage(page, addr);
@@ -1257,7 +1257,7 @@
 	 * Should we do an early C-O-W break?
 	 */
```

```
     if (write_access && !(vma->vm_flags & VM_SHARED)) {
-            struct page * page = alloc_page(GFP_HIGHUSER);
+            struct page * page = alloc_page(GFP_USER);  /* esdc: was GFP_HIGHUSER */
             if (!page) {
                     page_cache_release(new_page);
                     return -1;
diff -ruN linux/mm/page_alloc.c stage/mm/page_alloc.c
--- linux/mm/page_alloc.c    2002-02-25 12:38:14.000000000 -0700
+++ stage/mm/page_alloc.c    2004-01-26 01:05:02.639989000 -0700
@@ -56,6 +56,9 @@
   */
  #define BAD_RANGE(zone,x) (((zone) != (x)->zone) || (((x)-mem_map) <
(zone)->zone_start_mapnr) || (((x)-mem_map) >=
(zone)->zone_start_mapnr+(zone)->size))

+atomic_t esdc_nr_freed_pages = ATOMIC_INIT(0);
+atomic_t esdc_nr_allocd_pages = ATOMIC_INIT(0);
+
  /*
   * Buddy system. Hairy. You really aren't expected to understand this
   *
@@ -69,6 +72,7 @@
     free_area_t *area;
     struct page *base;
     zone_t *zone;
+    int i; /* esdc */

     /* Yes, think what happens when other parts of the kernel take
      * a reference to a page in order to pin it for io. -ben
@@ -111,6 +115,10 @@

     zone->free_pages -= mask;

+    if (zone->name[0] == 'H')  /* esdc: check first char of HighMem zone name */
+        for (i = 0; i < -mask; i++)
+            atomic_inc(&esdc_nr_freed_pages);
+
     while (mask + (1 << (MAX_ORDER-1))) {
         struct page *buddy1, *buddy2;

@@ -143,6 +151,8 @@
     return;

  local_freelist:
+    goto back_local_freelist; /* esdc: prevent single-page overflow */
+
     if (current->nr_local_pages)
         goto back_local_freelist;
     if (in_interrupt())
@@ -185,6 +195,7 @@
     struct list_head *head, *curr;
     unsigned long flags;
     struct page *page;
+    int i;  /* esdc */

     spin_lock_irqsave(&zone->lock, flags);
     do {
@@ -203,6 +214,11 @@
                 MARK_USED(index, curr_order, area);
             zone->free_pages -= 1UL << order;

+            /* esdc: check first char of HighMem zone name */
+            if (zone->name[0] == 'H')
+                for (i = 0; i < 1UL << order; i++)
+                    atomic_inc(&esdc_nr_allocd_pages);
+
        page = expand(zone, page, index, order, curr_order, area);
        spin_unlock_irqrestore(&zone->lock, flags);

@@ -231,6 +247,49 @@
 }
 #endif

+static long int esdc_prev_overflow = 0;  /* look for edges */
+
+/* esdc: monitor overflows or underflows without verbose output
+ * check for large changes in the current overflow state
+ */
+void esdc_overflow_info(zone_t *zone, int balance_classzone, int verbose)
+{
+   struct sysinfo i;  /* esdc */
+   long int esdc_real_cache_size;
+   long int esdc_curr_overflow;
+
+   si_meminfo(&i);
+
+   esdc_real_cache_size = atomic_read(&page_cache_size) - i.bufferram
+               - swapper_space.nrpages;
+
+   if (esdc_real_cache_size != (i.totalhigh - i.freehigh))
+   {
+       esdc_curr_overflow = esdc_real_cache_size -
+               (i.totalhigh - i.freehigh);
+       /* add threshold to reduce frequency of error messages */
+       if ((esdc_curr_overflow > (esdc_prev_overflow + 500)) ||
+           (esdc_curr_overflow < (esdc_prev_overflow - 500)))
+       {
+           if (balance_classzone)
+           printk("esdc: **** balance_classzone overflow **** ");
+           else
+           printk("esdc: **** shrink_caches overflow **** ");
+
+           printk("by %ld pages\n", esdc_curr_overflow);
+           printk("esdc: page cache %ld   freehighmem %ld  pid %d\n",
+           esdc_real_cache_size, i.freehigh, current->pid);
+           printk("esdc: zone name %s, zone size %d \n",
+               zone->name, zone->size);
+
+           esdc_prev_overflow = esdc_curr_overflow;
+
+           if (verbose) show_free_areas();
+       }
+
+   }
+}
+
 static struct page * FASTCALL(balance_classzone(zone_t *, unsigned int,
unsigned int, int *));
 static struct page * balance_classzone(zone_t * classzone, unsigned int
gfp_mask, unsigned int order, int * freed)
 {
@@ -249,6 +308,8 @@

    current->flags &= ~(PF_MEMALLOC | PF_FREE_PAGES);

+   goto out; /* esdc: repair single-page page cache overflow */
+
    if (current->nr_local_pages) {
        struct list_head * entry, * local_pages;
        struct page * tmp;
@@ -292,6 +353,7 @@
        nr_pages = current->nr_local_pages;
```

```
 		/* free in reverse order so that the global order will be lifo */
 		while ((entry = local_pages->prev) != local_pages) {
+			/* esdc: this section appears to never be used */
 			list_del(entry);
 			tmp = list_entry(entry, struct page, list);
 			__free_pages_ok(tmp, tmp->index);
@@ -301,10 +363,12 @@
 		current->nr_local_pages = 0;
 	}
  out:
+	esdc_overflow_info(classzone, 1, 0);
 	*freed = __freed;
 	return page;
 }
 
+
 /*
  * This is the 'heart' of the zoned buddy allocator:
  */
@@ -399,6 +463,8 @@
 	if (order > 3)
 		return NULL;
 
+	printk("esdc: __alloc_pages: calling schedule\n");
+
 	/* Yield for kswapd, and try again */
 	current->policy |= SCHED_YIELD;
 	__set_current_state(TASK_RUNNING);
@@ -610,6 +676,7 @@
 #endif
 					zonelist->zones[j++] = zone;
 				}
+				break; /* esdc: restrict ESDC to highmem zone */
 			case ZONE_NORMAL:
 				zone = pgdat->node_zones + ZONE_NORMAL;
 				if (zone->size)
diff -ruN linux/mm/swap_state.c stage/mm/swap_state.c
--- linux/mm/swap_state.c	2001-10-31 16:31:03.000000000 -0700
+++ stage/mm/swap_state.c	2003-08-05 11:54:10.859997000 -0600
@@ -200,7 +200,8 @@
 		 * Get a new page to read into from swap.
 		 */
 		if (!new_page) {
-			new_page = alloc_page(GFP_HIGHUSER);
+			/* retain GFP_HIGHUSER for esdc */
+			new_page = alloc_page(GFP_HIGHUSER);
 			if (!new_page)
 				break;		/* Out of memory */
 		}
diff -ruN linux/mm/swapfile.c stage/mm/swapfile.c
--- linux/mm/swapfile.c	2002-02-25 12:38:14.000000000 -0700
+++ stage/mm/swapfile.c	2003-11-13 23:24:35.509998000 -0700
@@ -17,6 +17,7 @@
 #include <linux/compiler.h>
 
 #include <asm/pgtable.h>
+atomic_t esdc_nr_will_swap	= ATOMIC_INIT(0);
 
 spinlock_t swaplock = SPIN_LOCK_UNLOCKED;
 unsigned int nr_swapfiles;
@@ -203,6 +204,7 @@
 			if (offset > p->highest_bit)
 				p->highest_bit = offset;
 			nr_swap_pages++;
+			atomic_inc(&esdc_nr_will_swap);
 		}
```

```
    }
    return count;
diff -ruN linux/mm/vmscan.c stage/mm/vmscan.c
--- linux/mm/vmscan.c     2002-02-25 12:38:14.000000000 -0700
+++ stage/mm/vmscan.c     2004-01-26 01:05:02.659991000 -0700
@@ -32,6 +32,13 @@
   */
 #define DEF_PRIORITY (6)

+/* esdc: atomic variables for analyzing shrink_cache */
+atomic_t esdc_nr_dirty          = ATOMIC_INIT(0);
+atomic_t esdc_nr_anonymous      = ATOMIC_INIT(0);
+atomic_t esdc_nr_still_cached = ATOMIC_INIT(0);
+atomic_t esdc_nr_drop_buffers = ATOMIC_INIT(0);
+atomic_t esdc_nr_remove_cache = ATOMIC_INIT(0);
+
 /*
  * The swap-out function returns 1 if it successfully
  * scanned all the pages it was asked to ('count').
@@ -132,7 +139,11 @@
        entry = get_swap_page();
        if (!entry.val)
            break;
-       /* Add it to the swap cache and mark it dirty
+
+       /* esdc: removed enhancement under development
+        * that copied anonymous page to high memory */
+
+       /* Add ito the swap cache and mark it dirty
         * (adding to the page cache will clear the dirty
         * and uptodate bits, so we need to do it again)
         */
@@ -402,6 +413,8 @@
            */
            int (*writepage)(struct page *);

+           /* printk("esdc: writepage called for dirty ESDC page\n"); */
+           atomic_inc(&esdc_nr_dirty);
            writepage = page->mapping->a_ops->writepage;
            if ((gfp_mask & __GFP_FS) && writepage) {
                ClearPageDirty(page);
@@ -443,6 +456,8 @@
                    /* effectively free the page here */
                    page_cache_release(page);

+                   atomic_inc(&esdc_nr_anonymous);
+
                    if (--nr_pages)
                        continue;
                    break;
@@ -454,6 +469,9 @@
                     */
                    page_cache_release(page);

+                   /* page to be freed later! */
+                   atomic_inc(&esdc_nr_still_cached);
+
                    spin_lock(&pagemap_lru_lock);
                }
            } else {
@@ -461,6 +479,8 @@
                UnlockPage(page);
                page_cache_release(page);

+               atomic_inc(&esdc_nr_drop_buffers);
+
                spin_lock(&pagemap_lru_lock);
                continue;
            }
@@ -477,6 +497,8 @@
 page_mapped:
            if (--max_mapped >= 0)
                continue;
+           /*printk("esdc: Inactive pages. Want to swap. pid: %d\n",
+               current->pid); */

            /*
             * Alert! We've found too many mapped pages on the
@@ -500,9 +522,11 @@
        /* point of no return */
        if (likely(!PageSwapCache(page))) {
            __remove_inode_page(page);
+           /* atomic_inc(&esdc_nr_remove_cache);   esdc */
            spin_unlock(&pagecache_lock);
        } else {
            swp_entry_t swap;
+           atomic_inc(&esdc_nr_remove_cache);  /* esdc */
            swap.val = page->index;
            __delete_from_swap_cache(page);
            spin_unlock(&pagecache_lock);
@@ -521,6 +545,8 @@
    }
    spin_unlock(&pagemap_lru_lock);

+   esdc_overflow_info(classzone, 0,0);
+
    return nr_pages;
 }

@@ -625,7 +651,8 @@
    int need_more_balance = 0, i;
    zone_t * zone;

-   for (i = pgdat->nr_zones -1; i >= 0; i--) {
+   /* esdc: changed from pgdat->nr_zones -1 to avoid balancing highmem */
+   for (i = pgdat->nr_zones -2; i >= 0; i--) {
        zone = pgdat->node_zones + i;
        if (unlikely(current->need_resched))
            schedule();
@@ -665,7 +692,8 @@
    zone_t * zone;
    int i;

-   for (i = pgdat->nr_zones -1; i >= 0; i--) {
+   /* esdc: changed from pgdat->nr_zones -1 to avoid balancing highmem */
+   for (i = pgdat->nr_zones -2; i >= 0; i--) {
        zone = pgdat->node_zones + i;
        if (!zone->need_balance)
            continue;
```

Appendix C

Experimental Automation Scripts

C.1 Remote Experimental Platform Support

The `sumo.pl` utility transfers all automation scripts, benchmark configuration files, benchmark applications, and kernel patches to the experimental Linux system. In this way, no files are edited on the experimental system, which prevents data loss in the event of a catastrophic system failure such as a corrupted file system.

Listing C.1: sumo.pl source

```
#!/usr/bin/perl
################################################################################
# sumo.pl
#
# Author:
#   John Koob
#
# Date:
#   2002/05/12
#
# Description:
#   Applies kernel patch to remote source tree.
#   Transfers benchmarks, configuration files, and
#   user scripts to remote host.
#
#   sumo - to choose, take up, apply, employ
#
# Usage:
#   see &Usage
#
################################################################################

use Getopt::Std;
use File::Basename;
use Carp;

local($work_path) = "$ENV{HOME}/thesis/work";
local($trees_path) = "$work_path/trees";
local($install_path) = "/usr/src/linux";
local($base_dir) = "linux";
local($dev_dir) = "dev";
local($stage_dir) = "stage";
local($project_dir);
local($project_path);
local($project_kernel) = "src/kernel";
local($kernel_tarball) = "kernels/linux-2.4.18.tar";
local($gnu_bin) = "/BOX/bin";
local($kernel_cpio) = "kernel.cpio";
local($other_cpio) = "other.cpio";
local($full_diff) = "full.diff";
local($incremental_diff) = "inc.diff";
local($diff_file);
local($dest_host) = "bart";
local($apply);
local($full);
local($incremental);
local($quick);
$0 = basename($0);
$| = 1;

&Usage(1) if (!&getopts('afim:p:qrh'));
$opt_h && &Usage(1);
&Usage(1) if (! $opt_p);

$apply = $opt_a;
$full = $opt_f;
$incremental = $opt_i;
$dest_host = $opt_m if $opt_m;
$project_dir = $opt_p;
$refresh = $opt_r;
$quick = $opt_q;

$project_path = "$work_path/$project_dir";
(-d "$project_path") ||
    carp("$0: No project: $project_path") && &Usage(1);

if ($refresh)
{
    print "Press enter if you really want to refresh: ";
    <STDIN>;
    &RefreshTree($dev_dir);
    &RefreshTree($stage_dir);
    &RefreshRemoteTree($install_path);
    print "Done.  So long for now.\n";
    exit;
}

if (-d "$project_path/$project_kernel")
{
    print "Staging kernel source...\n";
    chdir("$project_path/$project_kernel");
    `find . -name "CVS" -prune -o -print | cpio -oc > $trees_path/$kernel_cpio `;
    `cd $trees_path/$stage_dir; cat $trees_path/$kernel_cpio | cpio -i `;
}

print "Staging other source...\n";
chdir($project_path);
`find . -name "CVS" -prune -o -name "dat" -prune -o -name "app" -prune -o -name "
    kernel" -prune -o -print | cpio -oc > $trees_path/$other_cpio `;

chdir($trees_path);
if ($full && !$quick)
{
    print "Creating full diff ...";
    $diff_file = $full_diff;
    `$gnu_bin/diff -ruN $base_dir $stage_dir > $diff_file `;
    print "done\n";
}
```

```
elsif ($incremental && !$quick)
{
    print "Creating incremental diff...";
    $diff_file = $incremental_diff;
    `$gnu_bin/diff -ruN $dev_dir $stage_dir > $diff_file `;
    print "done\n";
}

if ($apply)
{
    if ($quick && -d "$project_path/$project_kernel")
    {
        print "Extracting cpio to $dev_dir tree...\n";
        `cd $trees_path/$dev_dir; cat $trees_path/$kernel_cpio | cpio -i `;
        print "done\n";
        print "Extracting cpio to remote tree...\n";
        `echo "put $kernel_cpio" | sftp $dest_host `;
        `ssh $dest_host 'cd $install_path;cat ~/$kernel_cpio | sudo cpio -i' `;
        print "done\n";
    }
    elsif (-d "$project_path/$project_kernel")
    {
        print "Applying patch to $dev_dir tree...\n";
        `cd $dev_dir; $gnu_bin/patch -u -N -p1  < ../$diff_file `;
        print "done\n";
        print "Applying patch to remote tree...\n";
        `echo "put $diff_file" | sftp $dest_host `;
        `ssh $dest_host 'cd $install_path;sudo patch -u -N -p1 < ~/$diff_file' `;
        print "done\n";
    }
    print "Extracting other cpio to remote tree...\n";
    `echo "put $other_cpio" | sftp $dest_host `;
    `ssh $dest_host 'cd $project_dir; cat ~/$other_cpio | sudo cpio -i' `;
    print "done\n";
}

exit(0);

################################################################################
# RefreshTree
#
# Desc:
#    Removes a tree and extracts kernel source from tarball
#
# Input:
#    $tree
#
# Output:
#
################################################################################
sub RefreshTree
{
    my($dir) = @_;

    print "Confirm refresh of $dir tree in $trees_path:";
    <STDIN>;
    chdir($trees_path);
    print "Refreshing $dir tree in $trees_path.";
    `cd $dir; rm -fr * `;
    print ".";
    `cd $dir; tar xvf $work_path/$kernel_tarball `;
    print ".";
    `mv $dir/linux $trees_path/tmp `;
    `rmdir $dir `;
    `mv $trees_path/tmp $trees_path/$dir `;
    print "done\n";
}

################################################################################
# RefreshRemoteTree
#
# Desc:
#    Removes a tree and extracts kernel source from tarball
#
# Input:
#    $tree
#
# Output:
#
################################################################################
sub RefreshRemoteTree
{
    my($dir) = @_;
    my($cmd);

    print "Must manually remove old tree first! \nConfirm refresh of $dir tree:";
    <STDIN>;
    print "Refreshing $dir tree...\n";
    $cmd = "cd $dir; sudo tar xvf /root/$kernel_tarball;";
    $cmd .= "sudo mv $dir/linux/* $dir; sudo rmdir linux";
    `ssh $dest_host '$cmd'`;
    print "done\n";
}

################################################################################
# Usage
#
# Desc:
#    Print usage statement.
#
# Input:
#    $ret_code    - program exit code
#
# Output:
#    exits with $ret_code
#
################################################################################
sub Usage
{
    my($ret_code) = @_;

    &Usage(1) if (!&getopts('afip:qrh'));
    print "\nUsage: $0 [-afih] [-p <project>]\n";
    print "where:\n";
    print "   -a           Apply patch\n";
    print "                \n";
    print "   -f           Full patch - against official kernel\n\n";
    print "   -i           Incremental patch\n";
    print "                \n";
    print "   -m           test machine name\n";
    print "                \n";
    print "   -h           This help\n";
    print "                \n";
    print "   -p           Name of project directory\n";
    print "                \n";
    print "   -q           Quick mode; do not install patch\n";
    print "                \n";
    print "   -r           Refresh dev and stage directories\n";
    print "                \n";

    exit $ret_code if ($ret_code);
} # Usage
```

C.2 Automation and Data Acquisition

The experiments are automated using the `mando.pl` utility. All experimental parameters are specified via an experimental command file. For example, the configuration of a PostMark experiment would appear as shown in Listing C.2.

Listing C.2: Experimental command file

```
# mando command file
#
# configuration section
# type parameter initial_value step_value cur_value axis
boot esdc 16M 16M 0M 1
boot dram 160M -16M 0M 2
esdc diag_str_len 0 0 0 0
esdc es_to_mem 300 0 300 0
esdc es_from_mem 300 0 300 0
esdc es_to_dev 300 0 300 0
esdc es_from_dev 300 0 300 0
esdc io_monitor 1 0
mtrr state uncachable
optn j 2 0 2 0
pass cycle 0 4 0
pass execx 0 7 0
pass execy 0 7 0
pass bypass 0 0 0
swap /dev/sda1 on
#
# benchmark command line
# ':' indicates initial and step values
postmark.pl -n 10000 -t 10000 -s 500,10004
# eof
```

The configuration section indicates that the x-axis is the *esdc* boot parameter. The initial value for this axis is 16 MB, which is incremented by 16 MB on each execution pass. Likewise, the y-axis is the main memory size that is calculated from the *esdc* and *mem* boot parameters. Here, it has an initial value of 160 MB, which is decremented by 16 MB on each execution pass. The number of execution passes are constrained by the *execx* and *execy* pass parameters. Various ESDC parameters specify the access time penalty of 300%. The caching property of the high memory zone is specified by the MTRR registers using the state parameter. Multiple swap devices can be enabled or disabled in this configuration file. The last non-commented line of the file contains the command line that will execute the experiment. If the *optn* parameter is an independent variable and matches a command line switch, the associated numeric value is adjusted before launching the experiment.

The `mando.pl` utility is a Perl script that automates the execution of multiple executions of an ESDC experiment. The primary benefit of this script is that it can coordinate multiple executions of an experiment to produce a comprehensive raw data file with up to two independent variables.

Listing C.3: mando.pl source

```
#!/usr/bin/perl
################################################################################
# mando.pl
#
# Author:
#   John Koob
#
# Date:
#   2003/04/30
#
# Description:
#   Linux benchmark automation
#   Runs a benchmark, records results, reboots and repeats for
#   different kernel configuration.
#
#   mando - to commit, entrust, order, command
#
# Usage:
#   see &Usage
#
################################################################################

use Getopt::Std;
use File::Basename;

local($esdc_home) = "$ENV{HOME}/esdc";
local($mando_suid) = "mando_suid.pl";
local($mando_start) = "mando_start.pl";
local($remote_host) = "barney";
local($remote_path) = "thesis/work/esdc/dat";
local($lilo_conf) = "/etc/lilo.conf";
local($proc_mtrr) = "/proc/mtrr";
local($proc_meminfo) = "/proc/meminfo";
local($proc_esdc) = "/proc/esdc";
local($proc_esdc_ctl) = "/proc/sys/vm/esdc_ctl";
local($tune_fsck) = "/sbin/tune2fs";
local($max_mtrrs) = 6;     # two of eight MTRR ranges used by default
local($cmd_file) = "";
local($out_file) = "";
local($verbose) = 1;       # turn off verbose with -q (quiet) option
local($esdc_pwd) = `pwd`; chomp($esdc_pwd);
local($passx) = 0;
local($passy) = 0;
local($bypass) = 0;
local($monitor) = 0;
$0 = basename($0);
$| = 1;


&Usage(1) if (!&getopts('c:o:x:y:bmqdh'));
$opt_h && &Usage(0);

&Usage(1) if (! $opt_c);
$cmd_file = $opt_c;

if (! -e "$cmd_file")
```

```
{
    &ReportError("$0: $cmd_file does not exist\n", $verbose);
}

&Usage(1) if (! $opt_o);
$out_file = $opt_o;

$passx = $opt_x;
$passy = $opt_y;

$bypass = $opt_b;

$monitor = $opt_m;

$verbose = 0 if $opt_q;

&TestAlignMemBlocks($passx, $passy) if ($opt_d);

&Mando($cmd_file, $out_file, $passx, $passy, $bypass, $monitor, $verbose);


exit(0);


###############################################################################
# NumSort
#
# Desc:
#     Numeric sort
#
###############################################################################
sub NumSort
{
    $a <=> $b;
}


###############################################################################
# GetMetricsFilename
#
# Desc:
#     Derive metrics filename from outfile
#
# Input:
#     $out_file
#
# Output:
#     $metrics_file
#
###############################################################################
sub GetMetricsFilename
{
    my($out_file) = @_;
    my($metrics_file);

    ($metrics_file = $out_file) =~ s/([^\.]+)\.\w*/$1.mtr/;

    return $metrics_file;
}

###############################################################################
# ParseCmdFile
#
# Desc:
# Parse command file
#     bench command line
#     type parameter initial value    step value
#
# Input:
#    $cmd_file
#    $params_ref
#    $bench_cmd_ref
#    $verbose
#
# Output:
#    $params_ref
#
##########################################################################
sub ParseCmdFile
{
    my($params_ref , $bench_cmd_ref , $cmd_file ,
        $passx , $passy , $bypass , $verbose ) = @_;
    my( @lines );
    my( $line );
    my( $type );
    my( $param );
    my( $values );

    (open(FH, "$cmd_file") && ( @lines=<FH>) && close(FH))
        || &ReportError("$0: Cannot open $cmd_file\n" , $verbose );

    chomp( @lines );

    # extract command line and remove comments
    do
    {
        $$bench_cmd_ref = pop( @lines );
    }
    while ( $$bench_cmd_ref =~ /^\s*#/);

    printf "\nParsing command file...\n\n";
    print "====---===============---====================--\n";
    print "type   param              values \n";
    print "====---===============---====================--\n";
    foreach $line ( @lines)
    {
        # prune out comments
        next if ( $line =~ /^\s*#/);

        # extract first two fields plus an array of values
        ($type , $param , @values ) = split(/\s+/, $line );

        # build the big hash
        $$params_ref{$type}{$param} = [ @values ];

        # cmd line override for execution pass number
        # change value read from command file if pass is non-zero
        $$params_ref{pass}{execx }[2] = $passx  if ( $passx );
        $$params_ref{pass}{execy }[2] = $passy  if ( $passy );

        # bypass override to allow an fsck with no subsequent experiment
        if ( $bypass)
        {
            $$params_ref{pass}{bypass }[2] = $bypass;
        }
        else
        {
            $$params_ref{pass}{bypass }[2] = 0;
        }


        printf "%s    %-14s    ", $type , $param;
        printf "[@{$$params_ref{$type}{$param}}]\n";

    }
    printf "\n";

}


################################################################################
# MonitorMetrics
#
# Desc:
# Fork a child process and monitor main memory and ESDC metrics
#
# Input:
#    $out_file
#    $params_ref
#    $bench_cmd_ref
#    $verbose
#
# Output:
#    $pid
#
################################################################################
sub MonitorMetrics
{
    my($params_ref, $out_file, $verbose) = @_;
    my($metrics_file);
    my($pid);

    $metrics_file = &GetMetricsFilename($out_file);

#    unlink $metrics_file if(-e "$metrics_file");
    MFORK: {
        if ($pid = fork)      # parent
        {
            # display message and return to launch the benchmark
            print "Launched metrics process: $pid\n";
            return $pid;
        }
        elsif (defined $pid) # child
        {
            print `echo "#------new execution pass------" >> $metrics_file`;
            # sleep 1900;
            while (1)
            {
                print `cat $proc_meminfo $proc_esdc >> $metrics_file`;
                sleep 1000;
            }
            exit(0);
        }
        elsif ($! =~ /No more process/)
        {
            print "Ran out of processes\n";
            sleep 5;
            redo MFORK;
        }
        else
        {
            die "$0: Can not fork child process\n";
        }
    }

}
```

```
###############################################################################
# Startup
#
# Desc:
#    Set startup command line for mando.pl in mando_start.pl
#
# Input:
#    $params_ref
#    $cmd_file
#    $out_file
#    $done
#    $will_bypass
#    $monitor
#    $verbose
#
# Output:
#
###############################################################################
sub Startup
{
    my($params_ref, $cmd_file, $out_file, $done, $will_bypass,
        $monitor, $verbose) = @_;
    my($passx);
    my($passy);
    my($cmd);

    $passx = $$params_ref{pass}{execx}[2];
    $passy = $$params_ref{pass}{execy}[2];

    # force an fsck check on next reboot
    if ($will_bypass)
    {
        $cmd = "\'$tune_fsck -c 1 -C 1 /dev/sda8\'";
        print "Running tune2fs—will run fsck after next reboot ...\n";
        print `$esdc_home/bin/$mando_suid $cmd`;
    }
    # tell fsck to not check the filesystem for the following series of reboots
    elsif ($$params_ref{pass}{bypass}[2] != 0)
    {
        $cmd = "\'$tune_fsck -c 0 -i 0 /dev/sda8\'";
        print "Running tune2fs—will not run fsck ...\n";
        print `$esdc_home/bin/$mando_suid $cmd`;
    }

    # the command line to launch ourself again
    $cmd = "$esdc_home/bin/$0 -c $cmd_file -o $out_file -x $passx -y $passy";
    $cmd .= " -b" if $will_bypass;
    $cmd .= " -m" if $monitor;

    print "Will run following command after reboot:\n";
    print "    $cmd\n";

    # update the mando_auto.pl script that gets run by a Linux startup script
    # after reboot
    open(FH, ">$esdc_home/bin/$mando_start")
        || &ReportError("$0: Cannot open $mando_start\n", $verbose);

    # if done, then an empty perl script will be run by the
    # Linux startup script.
    if (! $done)
    {
        print FH "#!/usr/bin/perl\n";
        print FH "chdir(\"$esdc_pwd\");\n";
        print FH "print \"mando_start sleeping...\\n\";\n";
        print FH "system(\"sleep 15; $cmd\");\n";
    }
    close(FH);
}

################################################################################
# Reboot
#
# Desc:
#    Set kernel command line parameters for next reboot
#
#    The value of the next execution pass is saved by passing it as
#    a command line parameter to mando.pl when it is run after reboot
#
# Input:
#    $params_ref
#    $cmd_file
#    $out_file
#    $pid            # pid of child process to kill when done
#    $verbose
#
# Output:
#    status
#
################################################################################
sub Reboot
{
    my($params_ref, $cmd_file, $out_file, $pid, $verbose) = @_;
    my($boot_param);
    my($param) = "";
    my($done) = 0;
    my($will_bypass) = 0;
    my($reboot);
    my($monitor);

    # for monitoring next reboot
    $monitor = 1 if ($pid);

    # increment y axis and reboot
    $$params_ref{pass}{execy}[2] += 1;
    $reboot = ".reboot y";

    if ($$params_ref{pass}{execy}[2] > $$params_ref{pass}{execy}[1])
    {
        # if max y count reached and we are not bypassing an experiment,
        # then set the next reboot to bypass
        if ($$params_ref{pass}{bypass}[2] == 0)
        {
            $will_bypass = 1;
            $$params_ref{pass}{execy}[2] -= 1;
            $reboot = ".fsck";
        }
        else
        {
            # reset y axis, increment x axis, and reboot
            $$params_ref{pass}{execy}[2] = $$params_ref{pass}{execy}[0];
            $$params_ref{pass}{execx}[2] += 1;
            $reboot = ".reboot x";

            if ($$params_ref{pass}{execx}[2] > $$params_ref{pass}{execx}[1])
            {
                # kill the child process
                kill("9", $pid);
                # Have we exceeded the maximum number of passes?
                # If so, return indicating done. Remember, second
                # field of array is max - 1

                $done = 1;

                # clear the startup wrapper script so we don't run forever
                &Startup($params_ref, $cmd_file, $out_file,
                    $done, $will_bypass, $monitor, $verbose);

                return 1;
            }
        }
    }

    # set the boot parameters for the next execution of mando
    $param = &ConfigBoot($params_ref, $verbose);

    # save these kernel boot parameters in /etc/lilo.conf
    (open(FH, "$lilo_conf") && (@lines=<FH>) && close(FH))
        || &ReportError("$0: Cannot open $lilo_conf\n", $verbose);

    open(FH, ">/tmp/lilo.tmp");
    foreach (@lines)
    {
        chomp;
        s/^(.*append=.*)$/        append="$param"/;
        print FH "$_\n";
    }
    close(FH);

    print `sudo cp -f /tmp/lilo.tmp $lilo_conf; rm /tmp/lilo.tmp`;
    print `sudo /sbin/lilo 2>&1`;

    &Startup($params_ref, $cmd_file, $out_file,
        $done, $will_bypass, $monitor, $verbose);

    `echo "$reboot" >> $out_file`;

    print "Shutting down ...\n";
    `sudo /sbin/shutdown -r now`;

    return 0;
}

##############################################################################
# UpdateParam
#
# Desc:
# The fourth element in the array is a numeric value, where:
#    if 0, param is a constant
#    if 1, param is an independent variable (x axis)
#    if 2, param is an independent variable (y axis)
#
# Input:
#    $params_ref
#    $type          - specified type field for parameter
#    $param         - specified parameter
#    $verbose
#
# Output:
#
##############################################################################
sub UpdateParam
```

```
{
    my($params_ref, $type, $param) = @_;
    my($axis) = 0;
    my($pass) = 0;
    my($init);
    my($step);

    # get the axis number
    $init = $$params_ref{$type}{$param}[0];
    $step = $$params_ref{$type}{$param}[1];
    $axis = $$params_ref{$type}{$param}[3];

    $pass = $$params_ref{pass}{execx}[2] if ($axis == 1);
    $pass = $$params_ref{pass}{execy}[2] if ($axis == 2);

    # set the "current value element" in the parameter array
    # calculation: initial value + step_value * current iterative step
    $$params_ref{$type}{$param}[2] = $init + $step * $pass;
}


###########################################################################
# AlignMemBlocks
#
# Desc:
#   Create several subblocks that are sized to satisfy MTRR
#   alignment restrictions: Each MTRR region's base address has to
#   be aligned on a size boundary. This is a recursive function.
#
#   Intel architecture has a limited the number of MTRR registers
#   so the goal of this function is to minimize the number of
#   subblocks
#
# Input:
#   $base - original base address
#   $size - original size (possibly not aligned to base address)
#   %subblocks_ref - sizes of each subblock in MTRR region
#           where the keys are the base addresses of the subblocks
#
# Output:
#
###########################################################################
sub AlignMemBlocks
{
    my($base, $size, $subblocks_ref, $test) = @_;
    my($b_aligned, $s_aligned);
    my($top_base, $top_size);
    my($bot_base, $bot_size);
    my($s);
    my($order);
    my($mask);


    # if we are already aligned, break recursion
    return if ($size == 0);

    # determine the order of the size and create an associated mask
    $s = $size;
    $order = 0;
    while ($s != 0)
    {
        $s = $s >> 1;
        $order++;
    }

    do {
        # try an aligned block smaller than the previous block
        $order--;
        $mask = ~0 << ($order);

        # find the largest aligned block that can fit within size,
        # where the size of new block is a power of two.
        $s_aligned = 1 << $order;

        # determine the aligned base address which must be equal to or
        # greater than $base
        $b_aligned = (($base + $s_aligned) & $mask);
        $b_aligned = $base if (($b_aligned % $base == 0) ||
            ($base % $s_aligned == 0));

        if ($test)
        {
            printf "aligned base:  %6d   aligned size:  %6d   base:     %6d",
            $b_aligned, $s_aligned, $base;
            print "\n";
        }
    }
    until ((($b_aligned + $s_aligned) <= ($base + $size)) &&
          ($b_aligned % $s_aligned == 0));

    $$subblocks_ref{$b_aligned} = $s_aligned;

    $bot_base = $base;
    $bot_size = $b_aligned - $base;
    $top_base = $b_aligned + $s_aligned;
    $top_size = $size - ($s_aligned + ($b_aligned - $base));

    # recurse for lower non-aligned subblock
    &AlignMemBlocks($bot_base, $bot_size, $subblocks_ref);

    if ($test)
    {
        printf "top block base: %6d   top block size: %6d \n",
            $top_base, $top_size;
        printf "original size:  %6d   b_aligned:     %6d   s_aligned: %6d\n\n",
            $size, $b_aligned, $s_aligned;
    }

    # recurse for upper non-aligned subblock
    &AlignMemBlocks($top_base, $top_size, $subblocks_ref);

}

###############################################################################
# TestAlignMemBlocks
#
# Desc:
#     Test the align mem blocks for sample data
#
###############################################################################
sub TestAlignMemBlocks
{
    my($base, $size) = @_;
    my(%subblocks);

    &AlignMemBlocks($base, $size, \%subblocks, 1);

    print "\nMTRR subblocks:\n";
    foreach (sort NumSort keys %subblocks)
    {
        printf "    Base: %6d    Size: %6d\n", $_, $subblocks{$_};
    }

    exit(0);
}


################################################################################
# ConfigPass
#
# Desc:
#    Verify that the execution pass is valid
#    Inital reboot before first official pass may be requested.
#    Format of parameters hash entry:
#         pass state min max cur
#    where
#         execx min - initial execution pass value for x axis
#         execx max - maximum number of reboot cycles (default: no boot, one pass)
#         execx cur - reboot immediately if -1 (default: no reboot)
#         execy min - initial execution pass value for y axis
#         execy max - maximum number of reboot cycles (default: no boot, one pass)
#         execy cur - current execution pass value
#         cycle min - inital number of benchmark cycles
#         cycle max - maximum number of benchmark cycles before reboot
#         cycle cur - current number of benchmark cycles
#
# Input:
#    $params_ref
#    $pid
#    $verbose
#
# Output:
#
################################################################################
sub ConfigPass
{
    my($params_ref, $pid, $verbose) = @_;

    # if bypass is set, do not allow any experiments to be run
    if ($$params_ref{pass}{bypass}[2] != 0)
    {
        return 0;
    }

    if ($$params_ref{pass}{cycle}[2] == 0)
    {
        print "Validating pass parameters ...\n";
        foreach $pass_param (sort keys %{$$params_ref{pass}})
        {
            &ReportError("$0: cmd file pass parameter out of range\n",
                $verbose)
                if (($$params_ref{pass}{$pass_param}[2] <
                $$params_ref{pass}{$pass_param}[0] - 1) ||
                ($$params_ref{pass}{$pass_param}[2] >
                $$params_ref{pass}{$pass_param}[1]));
        }

        # Now that we have all parameters, we may still need to reboot
        # to start experimental sequence
        if ($$params_ref{pass}{execx}[2] == -1)
        {
            printf "Initial pass will start after reboot ...\n";
            &Reboot(\%params, $cmd_file, $out_file, $pid, $verbose);
            exit(0); # does not make sense to fall through when testing
        }
    }
    else
    {
        print "Next execution cycle.\n";
    }

    if ($$params_ref{pass}{cycle}[2] <= $$params_ref{pass}{cycle}[1])
    {
        printf "  Current execution pass:  execx:%d  execy:%d  cycle:%d\n",
        $$params_ref{pass}{execx}[2],
        $$params_ref{pass}{execy}[2],
        $$params_ref{pass}{cycle}[2];
        return 1;
    }
    else
    {
        # return failure if maximum cycle count reached
        return 0;
    }

}


##############################################################################
# ConfigBoot
#
# Desc:
#   Get kernel command line memory parameters in units of MB
#   Format of parameters hash entry:
#       boot kernel_parameter init_value step_value cur_value
#
# Input:
#   $params_ref
#   $verbose
#
# Output:
#   kernel boot parameter string
#
##############################################################################
sub ConfigBoot
{
    my($params_ref, $verbose) = @_;
    my($boot_param);
    my($param_count) = 0;
    my($mem_val);
    my($esdc_val);
    my($dram_val);

    print "Setting boot parameters ...\n";
    foreach $boot_param (sort keys %{$$params_ref{boot}})
    {
        &UpdateParam($params_ref, "boot", $boot_param);

        $param_count++;

        # append units of MB that perl truncates during the above calculation
        $$params_ref{boot}{$boot_param}[2] .= "M";
    }
    &ReportError("$0: no boot parameters specified\n", $verbose)
        if (! $param_count);

    $esdc_val = $$params_ref{boot}{esdc}[2];
    $dram_val = $$params_ref{boot}{dram}[2];

    # the mem boot parameter is the total memory visible to the OS
    $mem_val = $$params_ref{boot}{dram}[2] + $$params_ref{boot}{esdc}[2];
    $mem_val .= "M";
    $boot_param_str = "esdc=$esdc_val mem=$mem_val";

    print "   dram=$dram_val $boot_param_str\n";

    return $boot_param_str;
}


###############################################################################
# ConfigSwap
#
# Desc:
#    Turn on or off swap devices
#    Format of parameters hash entry:
#        swap device on/off 0
#
# Input:
#    $params_ref
#    $verbose
#
# Output:
#
###############################################################################
sub ConfigSwap
{
    my($params_ref, $verbose) = @_;
    my($device);
    my($state);
    my($cmd);

    foreach $device (sort keys %{$$params_ref{swap}})
    {
        $state = $$params_ref{swap}{$device}[0];
        print "Setting swap device $device ...\n";

        $cmd = "\'swap$state -v $device 2>/dev/null\'";
        print "   ";
        print `$esdc_home/bin/$mando_suid $cmd`;
    }

    &ReportError("$0: cmd file requires a swap parameter\n", $verbose)
        if (!$state);

}


###############################################################################
# ConfigMtrr
#
# Desc:
#    Set caching properties of ESDC memory zone
#    Format of parameters hash entry:
#        mtrr state uncachable 0
#
# Input:
#    $params_ref
#    $verbose
#
# Output:
#
###############################################################################
sub ConfigMtrr
{
    my($params_ref, $verbose) = @_;
    my($state);
    my($mb_unit) = 2**20;
    my($base);
    my($size);
    my(%subblocks);
    my($i);
    my($cmd);

    # MTRR needs only one property set so no loop is necessary
    $state = $$params_ref{mtrr}{state}[0];

    &ReportError("$0: cmd file requires an mtrr parameter\n", $verbose)
        if (!$state);

    # need to calculate using original boot parameters again
    &ConfigBoot($params_ref, $verbose);

    print "Setting $$params_ref{boot}{esdc}[2] MB of ESDC as $state ...\n";

    # default memory caching property is write-back so nothing
    # needs to be done if this is the requested setting
    if ($state =~ /write-back/)
    {
        print "  No entry added. Default caching property is write-back\n";
        return;
    }

    # determine the number and size of subblocks if the base of
    # the MTRR region was not aligned to size of the entire region
    &AlignMemBlocks($$params_ref{boot}{dram}[2], $$params_ref{boot}{esdc}[2],
        \%subblocks, 0);

    &ReportError("$0: too many MTRR registers required\n", $verbose)
        if ((keys %subblocks) > $max_mtrrs);

    foreach (sort NumSort keys %subblocks)
    {
        printf "   base: %6d   size: %6d\n", $_, $subblocks{$_};
    }

    # multiple MTRR regions are needed if base was not originally aligned
    foreach (sort NumSort keys %subblocks)
    {
        $base = sprintf "%lx", ($_ * $mb_unit);
        $size = sprintf "%lx", ($subblocks{$_} * $mb_unit);
        $cmd =
        "   \'echo \"base=0x$base size=0x$size type=$state\" > $proc_mtrr\'";

        print "$cmd\n";
        print '$esdc_home/bin/$mando_suid $cmd 2>&1';
    }

}

##########################################################################
# ConfigEsdc
#
# Desc:
#   Set ESDC memory behavior
#   Format of parameters hash entry:
#       esdc  parameter  default_percentage  step  current_percentage
#
# Input:
#   $params_ref
#   $verbose
#
```

```
# Output:
#
################################################################################
sub ConfigEsdc
{
    my($params_ref, $verbose) = @_;
    my($esdc_param);
    my($ctl_line) = "";
    my($cmd);

    # assume that all esdc parameters support incrementals,
    # even if the step value is not applicable for a given
    # parameter
    foreach $esdc_param (sort keys %{$$params_ref{esdc}})
    {
        &UpdateParam($params_ref, "esdc", $esdc_param);
    }

    # if additional columns are present in proc file but explicitly
    # set here, the command still is successful
    $ctl_line .= "$$params_ref{esdc}{es_to_mem}[2]    ";
    $ctl_line .= "$$params_ref{esdc}{es_from_mem}[2]    ";
    $ctl_line .= "$$params_ref{esdc}{es_to_dev}[2]    ";
    $ctl_line .= "$$params_ref{esdc}{es_from_dev}[2]    ";
    $ctl_line .= "$$params_ref{esdc}{diag_str_len}[2]    ";
    $ctl_line .= "$$params_ref{esdc}{io_monitor}[2]    ";

    print "Setting ESDC configuration parameters ...\n";
    $cmd = "    \'echo \"$ctl_line\" > $proc_esdc_ctl\'";
    print "$cmd\n";
    print `$esdc_home/bin/$mando_suid $cmd`;

}


################################################################################
# ConfigOptn
#
# Desc:
#    Update all variable benchmark command line option values
#
# Input:
#    $params_ref
#    $bench_cmd_ref
#    $verbose
#
# Output:
#
################################################################################
sub ConfigOptn
{
    my($params_ref, $bench_cmd_ref, $verbose) = @_;
    my($optn_param);
    my($cur_value);


    print "Updating benchmark command line options ...\n";
    print "    ";
    foreach $optn_param (sort keys %{$$params_ref{optn}})
    {
        &UpdateParam($params_ref, "optn", $optn_param);

        $cur_value = $$params_ref{optn}{$optn_param}[2];

        $$bench_cmd_ref =~ s/(\s+\-\-?$optn_param\s+)(\S+)/$1$cur_value/;
```

```
            print "$optn_param = $cur_value  ";
        }
}


################################################################################
# LaunchBenchmark
#
# Desc:
#    Calculate benchmark command line parameter values taking into
#    account the value of the current execution pass, if appropriate.
#
#    One or more arguments on the benchmark command line *may* be
#    altered with step values appended to the initial value
#    delimited by a ':'. This expression will be converted to
#    the number n where n = i + s * (current pass).
#
# Input:
#    $params_ref
#    $bench_cmd_ref
#    $verbose
#
# Output:
#
################################################################################
sub LaunchBenchmark
{
    my($params_ref, $bench_cmd, $out_file, $verbose) = @_;
    my($esdc_param);
    my($init);
    my($step);
    my($n);
    my($remote_node);
    my($metrics_file);

    if (! -e "$out_file")
    {
        `touch $out_file`;
    }

    open(FH, ">>$out_file");
    print FH ".param\n";
    print FH "#============================================\n";
    print FH "# type    param              values \n";
    print FH "#============================================\n";
    foreach $type (sort keys %{$params_ref})
    {
        foreach $esdc_param (sort keys %{$$params_ref{$type}})
        {
            printf FH "  %s   %-14s   ", $type, $esdc_param;
            printf FH "[@{$$params_ref{$type}{$esdc_param}}]\n";
        }
    }
    print FH "#-------------------------------------------\n";
    print FH ".bench $bench_cmd\n";
    close(FH);

    # Dump raw benchmark output to $out_file and then
    # the plotting script will have fcns to
    # parse the raw benchmark output.
    print "\nRunning benchmark ...\n";
    print "   $bench_cmd >> $out_file\n";
    print "----------------------------------------------------------\n";
    `$esdc_home/bin/$bench_cmd >> $out_file\n`;
    `echo "" >> $out_file\n`;
    print "----------------------------------------------------------------------\n";

    # Send the output file to remote host at the end of each run.
    # This facilitates remote experimental monitoring.
    ($remote_node) = ($bench_cmd =~ /\s*([^\s\.]+)/);
    print `scp $out_file $remote_host:$remote_path/$remote_node`;

    # If we are monitoring ourselves, also send back the monitoring data
    $metrics_file = &GetMetricsFilename($out_file);
    if (-e $metrics_file)
    {
        print `scp $metrics_file $remote_host:$remote_path/$remote_node`;
    }

    # Update the current benchmark cycle count
    $$params_ref{pass}{cycle}[2] += 1;

}


################################################################################
# Mando
#
# Desc:
# Parse command file
#     bench_cmd= bench command line
#     type parameter initial value   step value
# Configure system for ESDC experiments
#     ConfigPass - verify that the execution pass is valid
#     ConfigBoot - set visible memory size and ESDC size from boot parameters
#     ConfigSwap - set swap devices on or off
#     ConfigMtrr - set caching properties of ESDC using MTRR registers
#     ConfigEsdc - set ESDC penalties and miscellaneous configuration
#     ConfigOptn - set incremental option values, if any
# Run Benchmark
#     use benchmark parameters in benchmark command line
#
# Input:
#    $cmd_file
#    $out_file
#    $pass
#    $verbose
#
# Output:
#    @users
#
################################################################################
sub Mando
{
    my($cmd_file, $out_file, $passx, $passy, $bypass, $monitor, $verbose) = @_;
    my(%params);
    my($bench_cmd);
    my($cycles);
    my($monitor_pid) = 0;

    # Read in configuration parameters and the benchmark command line
    &ParseCmdFile(\%params, \$bench_cmd, $cmd_file,
        $passx, $passy, $bypass, $verbose);

    # Launch a subprocess to monitor memory and esdc metrics
    $monitor_pid = &MonitorMetrics(\%params, $out_file, $verbose) if ($monitor);

    # Verify current execution pass number and execution cycles
    while (&ConfigPass(\%params, $monitor_pid, $verbose))
    {
        # Config swap device(s)
        &ConfigSwap(\%params, $verbose);

        # Config esdc memory zone properties
        &ConfigMtrr(\%params, $verbose);

        # Config esdc properties
        &ConfigEsdc(\%params, $verbose);

        # Config independent variables
        &ConfigOptn(\%params, \$bench_cmd, $verbose);

        # Run benchmark
        &LaunchBenchmark(\%params, $bench_cmd, $out_file, $verbose);
    }


    if (&Reboot(\%params, $cmd_file, $out_file, $monitor_pid, $verbose))
    {
        # wrap up
        print "Done.\n";
    }

}

################################################################################
# ReportError
#
# Desc:
#   Print error message.
#
# Input:
#   $message    - error message
#   $display    - flag to control display of error message
#
# Output:
#   none
#
################################################################################
sub ReportError
{
        my($message, $display) = @_;

        print STDERR $message if ($display);
        exit(1);
} # ReportError


################################################################################
# Usage
#
# Desc:
#   Print usage statement.
#
# Input:
#   $ret_code    - program exit code
#
# Output:
#   exits with $ret_code
#
################################################################################
sub Usage
{
        my($ret_code) = @_;
        print "\nUsage: $0 -c <cmd_file> -o <out_file> -x <passx> -y <passy>";
        print "[-m] [-q] [-d] [-h]\n";
        print "\nwhere\n";
        print "    -c command file\n";
        print "    -o output file\n";
        print "    -x pass number override for x axis\n";
        print "    -y pass number override for y axis\n";
        print "    -b bypass experiment for current reboot to allow an fsck\n";
        print "    -m spawn a process to monitor memory and esdc metrics\n";
        print "    -q quiet\n";
        print "    -d debug\n";
        print "    -h this help\n";
        print "    \n";
        print "    \n";

        exit $ret_code;
} # Usage
```

C.3 Experimental Data Visualization

The `dispono.pl` utility processes the raw data file generated by `mando.pl`. With visualization options specified in the command line, `dispono.pl` generates a data file that is suitable for plotting. GNUplot is launched automatically to produce an encapsulated PostScript plot of the experimental results.

Listing C.4: dispono.pl source

```perl
#!/usr/bin/perl
################################################################################
# dispono.pl
#
# Author:
#   John Koob
#
# Date:
#   2003/04/30
#
# Description:
#   Generic facility for ESDC plot generation for any supported benchmark.
#   Converts a *.raw file to one or more *.dat files in a format
#   suitable for gnuplot.
#
#   dispono - to arrange, put in order, draw up
#
# Usage:
#   see &Usage
#
################################################################################

use Getopt::Std;
use File::Basename;

local($esdc_home) = "$ENV{HOME}/esdc";
local($gnuplot) = "/BOX/bin/gnuplot";
local($dispono_suid) = "dispono_suid.pl";
local($dispono_start) = "dispono_start.pl";
local($raw_file) = "";
local($dat_file) = "";
local($verbose) = 1;        # turn off verbose with -q (quiet) option
local($series) = 0;
local($bisect) = 0;
local($xrange) = 0;
local($yrange) = 0;
local($zrange) = 0;
local($errorbars) = 0;
local($confbars) = 0;
local($family) = ();
local($label) = ();
local(%gconfig);
local($esdc_pwd) = `pwd`; chomp($esdc_pwd);
$0 = basename($0);


&Usage(1) if (!&getopts('r:d:s:b:x:y:z:f:l:ecqh'));
$opt_h && &Usage(0);

$target_dir = $opt_d if ($opt_d);

&Usage(1) if (! $opt_r);
```

```
$raw_file = $opt_r;

&Usage(1) if (! $opt_d);
$dat_file = $opt_d;

$series = $opt_s if $opt_s;
$errorbars = $opt_e if $opt_e;
$confbars = $opt_c if $opt_c;
$bisect = $opt_b if $opt_b;
$xrange = $opt_x if $opt_x;
$yrange = $opt_y if $opt_y;
$zrange = $opt_z if $opt_z;
$family = $opt_f if $opt_f;
$label = $opt_l if $opt_l;
$verbose = 0 if $opt_q;

&GraphConfig(\%gconfig, $series, $bisect, $xrange, $yrange, $zrange,
$family, $label, $errorbars, $confbars, $verbose);

&Dispono($raw_file, $dat_file, \%gconfig, $verbose);

exit(0);


################################################################################
# ParseRawFile
#
# Desc:
# Parse raw benchmark output file
#
# Format of pdata structure
#     $pdata_ref[$x][$y][$c]{$type}{$param}[@values]
#   where $type is a parameter type from the parameter block, or
#         $type is {text} for benchmark text
#         $param is {raw} for raw benchmark text, or
#         $param is {data} for filtered benchmark data, or
#         $param is {bench} for benchmark command line
#
# Format of parameter block
#     type parameter initial value   step value  current value
#
# Input:
#    $pdata_ref
#    $bench_cmd_ref
#    $raw_file
#    $verbose
#
# Output:
#    $pdata_ref
#
################################################################################
sub ParseRawFile
{
    my($pdata_ref, $bench_cmd_ref, $raw_file, $verbose) = @_;
    my(@lines);
    my($line);
    my($type);
    my($param);
    my($values);
    my($prune) = 1;     # prune initial directives
    my($param_block);   # set if within param block
    my($reboot_axis);   # axis that was incremented upon reboot
    my($x) = 0;         # execution pass for x axis
    my($y) = 0;         # execution pass for y axis
    my($c) = 0;         # benchmark cycle number
```

```
(open(FH, "$raw_file") && (@lines=<FH>) && close(FH))
    || &ReportError("$0: Cannot open $raw_file\n", $verbose);

chomp(@lines);

printf "\nParsing raw file...\n\n";
foreach $line (@lines)
{
    next if ($line =~ /^\s*#/);
    next if ($line =~ /^\s*.fsck/);

    # directives indicate the start of sections
    if ($line =~ /^\s*\.param/)              # start of param block
    {
        $param_block = 1;
        if ($prune) # ignore directive for first .param near start of file
        {
            $prune = 0;
            next;
        }
        $c++;
        next;
    }
    elsif ($line =~ /^\s*\.bench\s+(.*)$/) # start of benchmark output
    {
        $param_block = 0;
        $$bench_cmd_ref = $1;
        next;
    }
    elsif ($line =~ /^\s*\.reboot\s+(\w)/)        # start of boot block
    {
        $reboot_axis = $1;
        next if ($prune); # ignore directive if .reboot at top of file

        # increment y if y axis changed on reboot
        $y++ if ($reboot_axis =~ /y/);

        # increment x and reset y if x axis changed on reboot
        if ($reboot_axis =~ /x/)
        {
            $x++;
            $y = 0;
        }

        # reset cycle count for start of next execution cycle
        $c = 0;
        $prune = 1;
        next;
    }

    # save benchmark cmd line
    if (! @{$pdata_ref->[$x][$y][$c]{text}{bench}})
    {
        push(@{$pdata_ref->[$x][$y][$c]{text}{bench}}, $$bench_cmd_ref);
        print "bench[$x][$y][$c]: @{$pdata_ref->[$x][$y][$c]{text}{bench}}"
            if ($verbose);
        print "\n" if ($verbose);
    }

    # parse parameter block
    if ($param_block)
    {
        $line =~ s/\[//;      # toss left bracket
        $line =~ s/\]//;      # toss right bracket
        $line =~ s/^\s*//;    # toss leading whitespace
```

```
            # extract first two fields plus an array of values
            ($type, $param, @values) = split(/\s+/, $line);

            # build the big hash
            $pdata_ref->[$x][$y][$c]{$type}{$param} = [ @values ];

            printf ("%s   %-14s   ", $type, $param) if ($verbose);
            printf "[@{$pdata_ref->[$x][$y][$c]{$type}{$param}}]\n"
                if ($verbose);
        }
        else  # parse benchmark block
        {
            push(@{$pdata_ref->[$x][$y][$c]{text}{raw}}, $line);
        }

    } # foreach $line


}

################################################################################
# PrettyAxisLabel
#
# Desc:
#   Convert param tag to axis label
#
# Input:
#   $param
#   $axis_label_ref
#   $verbose
#
# Output:
#
################################################################################
sub PrettyAxisLabel
{
    my($param, $axis_label_ref, $verbose) = @_;
    my(%label_map);

    %label_map = (
        esdc        => 'ESDC (MB)',
        dram        => 'Main Memory (MB)',
        es_to_mem   => 'Access Time Penalty (%)       ',
        es_from_mem => 'Access Time Penalty (%)       ',
        es_to_dev   => 'Access Time Penalty (%)       ',
        es_from_dev => 'Access Time Penalty (%)       ',
    );

    $$axis_label_ref = $label_map{$param};
}

################################################################################
# ParseBonnieText
#
# Desc:
# Filter raw bonnie benchmark text block
#
# Input:
#   $pdata_ref
#   $dat_file
#   $verbose
#
# Output:
#
```

```
############################################################################
sub ParseBonnieText
{
    my($pdata_ref, $dat_file, $verbose) = @_;
    my($x);
    my($y);
    my($c);
    my($line);


    for ($x = 0; $x < @{$pdata_ref}; $x++)
    {
        for ($y = 0; $y < @{$pdata_ref->[$x]}; $y++)
        {
            for ($c = 0; $c < @{$pdata_ref->[$x][$y]}; $c++)
            {
                foreach $line (@{$pdata_ref->[$x][$y][$c]{text}{raw}})
                {
                    next if ($line =~ /(sec|---)/);
                    next if ($line =~ /^\s*$/);
                    $line =~ s/^\s+//;
                    # add raw data values if present
                    $pdata_ref->[$x][$y][$c]{text}{data} =
                        [ split (/\s+/,$line) ] if ($line);
                    if ($verbose)
                    {
                        print "bonnie[$x][$y][$c]: ";
                        print "@{$pdata_ref->[$x][$y][$c]{text}{data}}";
                        print "\n";
                    }
                }
            }
        }
    }

    $pdata_ref->[0][0][0]{text}{labels} = [
        ("Size", "CharOutput", "Char%CPU",
        "BlockOutput", "Block%CPU",
        "Rewrite", "Rewrite%CPU",
        "CharInput", "Char%CPU",
        "BlockInput", "Block%CPU",
        "RndSeeks", "RndSeeks%CPU") ];

    $pdata_ref->[0][0][0]{text}{units} = [
        ("MB", "KB/s", " ",
        "KB/s", " ",
        "KB/s", " ",
        "KB/s", " ",
        "KB/s", " ",
        "/s", " ") ];


}

############################################################################
# ParseBonniePPText
#
# Desc:
# Filter raw bonnie benchmark text block
#
# Input:
#    $pdata_ref
#    $dat_file
#    $verbose
#
```

```
# Output:
#
##############################################################################
sub ParseBonniePPText
{
    my($pdata_ref, $dat_file, $verbose) = @_;
    my($x);
    my($y);
    my($c);
    my($line);


    for ($x = 0; $x < @{$pdata_ref}; $x++)
    {
        for ($y = 0; $y < @{$pdata_ref->[$x]}; $y++)
        {
            for ($c = 0; $c < @{$pdata_ref->[$x][$y]}; $c++)
            {
                foreach $line (@{$pdata_ref->[$x][$y][$c]{text}{raw}})
                {
                    next if ($line =~ /(sec|--)/);
                    next if ($line =~ /^\s*$/);
                    next if ($line =~ /\s+/);
                    $line =~ s/^\w+,//;
                    # add raw data values if present
                    $pdata_ref->[$x][$y][$c]{text}{data} =
                        [ split (/,/, $line) ] if ($line);
                    if ($verbose)
                    {
                        print "bonnie[$x][$y][$c]: ";
                        print "@{$pdata_ref->[$x][$y][$c]{text}{data}}";
                        print "\n";
                    }
                }
            }
        }
    }

    $pdata_ref->[0][0][0]{text}{labels} = [
        ("Size", "CharOutput", "Char%CPU",
        "BlockOutput", "Block%CPU",
        "Rewrite", "Rewrite%CPU",
        "CharInput", "Char%CPU",
        "BlockInput", "Block%CPU",
        "RndSeeks", "RndSeeks%CPU") ];

    $pdata_ref->[0][0][0]{text}{units} = [
        ("MB", "KB/s", " ",
        "KB/s", " ",
        "KB/s", " ",
        "KB/s", " ",
        "KB/s", " ",
        "/s", " ") ];


}

##############################################################################
# ConvertTimeOutput
#
# Desc:
# Convert output of /usr/bin/time into array of data values
#
# Input:
#   $line - line with /usr/bin/time output
```

```
#
# Output:
#    $time_data_ref - reference to array of values:
#    elapsed user system rss major_faults minor_faults swaps
#
################################################################################
sub ConvertTimeOutput
{
    my($line, $time_data_ref) = @_;
    my(@data);

    $line =~ s/[^\d\.\s]//g;

    @data = split(/\s+/, $line);

    push(@$time_data_ref, @data);

}


################################################################################
# ParsePostmarkText
#
# Desc:
# Filter raw postmark benchmark text block
#
# Input:
#    $pdata_ref
#    $dat_file
#    $verbose
#
# Output:
#
################################################################################
sub ParsePostmarkText
{
    my($pdata_ref, $dat_file, $verbose) = @_;
    my($x);
    my($y);
    my($c);
    my($line);
    my(@time_data);
    my($quantity);
    my($i);


    for ($x = 0; $x < @{$pdata_ref}; $x++)
    {
        for ($y = 0; $y < @{$pdata_ref->[$x]}; $y++)
        {
            for ($c = 0; $c < @{$pdata_ref->[$x][$y]}; $c++)
            {
                foreach $line (@{$pdata_ref->[$x][$y][$c]{text}{raw}})
                {
                    # add raw data values if present
                    next if (! @{$pdata_ref->[$x][$y][$c]{text}{raw}});

                    if ($line =~ /^\d/)
                    {
                        @time_data = split(/ /, $line);
                        $i = 0;
                        foreach $quantity (@time_data)
                        {
                            $time_data[$i] /= 1000 if ($quantity > 50000);
                            $i++;
                        }
                        next;
                    }

                    &ConvertTimeOutput($line , \@time_data);

                    $pdata_ref->[$x][$y][$c]{text}{data} = [ @time_data ];
                    if ($verbose)
                    {
                        print "postmark[$x][$y][$c]: ";
                        print "@{$pdata_ref->[$x][$y][$c]{text}{data}}";
                        print "\n";
                    }
                }
            }
        }
    }

    $pdata_ref->[0][0][0]{text}{labels} = [
        ("TimeTotal" , "TimeTransaction" , "TransactionRate" ,
        "FileCreationRate" , "FileCreationAloneRate" , "FileCreationMixedRate" ,
        "FileReadRate" , "FileAppendRate" ,
        "FileDeletionRate" , "FileDeletionAloneRate" , "FileDeletionMixedRate" ,
        "DataReadRate" , "DataWriteRate" ,
        "Real" , "System" , "User" ,
        "RSS" , "MajorFaults" ,
        "MinorFaults" , "NumSwaps" ) ];

    $pdata_ref->[0][0][0]{text}{units} = [
        ("s" , "s" , "/s" ,
        "files/s" , "files/s" , "files/s" ,
        "files/s" ,"files/s" ,
        "files/s" , "files/s" , "files/s" ,
        "KB/s" , "KB/s" ,
        "s" , "s" , "s" ,
        "KB" , " " ,
        " " , " " ) ];


}

#################################################################################
# ParseLbuildText
#
# Desc:
# Filter raw lbuild application text block
#
# Input:
#    $pdata_ref
#    $dat_file
#    $verbose
#
# Output:
#
#################################################################################
sub ParseLbuildText
{
    my($pdata_ref , $dat_file , $verbose) = @_;
    my($x);
    my($y);
    my($c);
    my($line);
    my(@time_data);
    for ($x = 0; $x < @{$pdata_ref}; $x++)
    {
        for ($y = 0; $y < @{$pdata_ref->[$x]}; $y++)
        {
            for ($c = 0; $c < @{$pdata_ref->[$x][$y]}; $c++)
            {
                foreach $line (@{$pdata_ref->[$x][$y][$c]{text}{raw}})
                {
                    # add raw data values if present
                    next if (! $line);

                    &ConvertTimeOutput($line, \@time_data);

                    $pdata_ref->[$x][$y][$c]{text}{data} = [ @time_data ];
                    if ($verbose)
                    {
                        print "lbuild[$x][$y][$c]: ";
                        print "@{$pdata_ref->[$x][$y][$c]{text}{data}}";
                        print "\n";
                    }
                }
                @time_data = ();
            }
        }
    }

    $pdata_ref->[0][0][0]{text}{labels} = [
        ("Real", "User", "System",
        "RSS", "MajorFaults",
        "MinorFaults", "NumSwaps") ];

    $pdata_ref->[0][0][0]{text}{units} = [
        ("s", "s", "s",
        "KB", " ",
        " ", " ") ];


}


##############################################################################
# ParseMummerText
#
# Desc:
# Filter raw mummer application text block
#
# Input:
#    $pdata_ref
#    $dat_file
#    $verbose
#
# Output:
#
##############################################################################
sub ParseMummerText
{
    my($pdata_ref, $dat_file, $verbose) = @_;
    my($x);
    my($y);
    my($c);
    my($line);
    my(@data);


    for ($x = 0; $x < @{$pdata_ref}; $x++)
    {
        for ($y = 0; $y < @{$pdata_ref->[$x]}; $y++)
        {
            for ($c = 0; $c < @{$pdata_ref->[$x][$y]}; $c++)
            {
                foreach $line (@{$pdata_ref->[$x][$y][$c]{text}{raw}})
                {
                    # add raw data values if present
                    next if (! @{$pdata_ref->[$x][$y][$c]{text}{raw}});

                    if (($line =~ /# CONSTRUCTIONTIME/) ||
                        ($line =~ /# SPACE/))
                    {
                        $line =~ /(\d+\.\d+)$/;
                        push (@data, $1);
                        next;
                    }

                    &ConvertTimeOutput($line, \@data);

                    $pdata_ref->[$x][$y][$c]{text}{data} = [ @data ];
                    if ($verbose)
                    {
                        print "lbuild[$x][$y][$c]: ";
                        print "@{$pdata_ref->[$x][$y][$c]{text}{data}}";
                        print "\n";
                    }
                }
            }
        }
    }

    $pdata_ref->[0][0][0]{text}{labels} = [
        ("ConstTime", "Space", "Real", "System", "User",
        "RSS", "MajorFaults",
        "MinorFaults", "NumSwaps") ];

    $pdata_ref->[0][0][0]{text}{units} = [
        ("s", "MB", "s", "s", "s",
        "KB", " ",
        " ", " ") ];

}

################################################################################
# ParseBenchmarkText
#
# Desc:
# Filter raw benchmark text blocks and populate the pdata structure
#
# Format of pdata structure
#     $pdata_ref[$x][$y][$c]{$type}{$param}[@values]
#   where $type is a parameter type from the parameter block, or
#         $type is {text} for benchmark text
#         $param is {raw} for raw benchmark text, or
#         $param is {data} for filtered benchmark data, or
#         $param is {bench} for benchmark command line
#
# Format of parameter block
#     type parameter initial value   step value   current value
#
# Input:
#    $pdata_ref
#    $bench_cmd_ref
#    $dat_file
#    $verbose
```

```
#
# Output:
#     $pdata_ref
#
################################################################################
sub ParseBenchmarkText
{
    my( $pdata_ref , $bench_cmd_ref , $dat_file , $verbose ) = @_;
    my( @lines );
    my( $line );

    # parse benchmark command
    ($cmd) = ( $$bench_cmd_ref =~ /^\s*(\S+).*$/);

    if ($cmd =~ /^bonnie$/)
    {
        &ParseBonnieText( $pdata_ref , $dat_file , $verbose );
    }

    if ($cmd =~ /^bonnie\+\+/)
    {
        &ParseBonniePPText( $pdata_ref , $dat_file , $verbose );
    }

    if ($cmd =~ /^postmark/)
    {
        &ParsePostmarkText( $pdata_ref , $dat_file , $verbose );
    }

    if ($cmd =~ /^lbuild/)
    {
        &ParseLbuildText( $pdata_ref , $dat_file , $verbose );
    }

    if ($cmd =~ /^mummer/)
    {
        &ParseMummerText( $pdata_ref , $dat_file , $verbose );
    }

    # add more benchmarks here
}

################################################################################
# IsNextIndex
#
# Desc:
#     Determine if we are at the start of an index
#
# Input:
#     $data_ref
#     $verbose
#
# Output:
#     $status
#
################################################################################
sub IsNextIndex
{
    my( $data_ref , $verbose )=@_;
    my( $index_delimiter );

    $index_delimiter = substr( $$data_ref , -2);

    if (( $index_delimiter =~ /\n\n/) || (! $index_delimiter))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}


###############################################################################
# ConfidenceInterval
#
# Desc:
#   Find the 95% confidence interval for a specified datapoint
#
# Input:
#   $pdata_ref
#   $s              data series
#   $x              current x value
#   $y              current y value
#   $verbose
#
# Output:
#   $min_ci_ref       reference to array of lower C.I. error bar points
#   $max_ci_ref       reference to array of upper C.I. error bar points
#
###############################################################################
sub ConfidenceInterval
{
    my($pdata_ref, $mean, $s, $x, $y, $min_ci_ref, $max_ci_ref, $verbose)=@_;
    my($c);
    my($n);
    my($cur_data);
    my($sum_squares) = 0;
    my($sigma);
    my($std_err);
    my($z) = 1.96;

    $n = @{$pdata_ref->[$x][$y]};

    # Run through all cycles for the current execution pass
    # That is, loop through all elements in the data line
    for ($c = 0; $c < $n; $c++)
    {
        $cur_data = $pdata_ref->[$x][$y][$c]{text}{data}[$s];

        # calculate ($cur_data - $mean)^2
        $sum_squares += ($cur_data - $mean)**2;
    }

    # calculate the standard deviation
    $sigma = sqrt($sum_squares / ($n - 1));

    # estimate the standard error
    $std_err = $sigma / sqrt($n);

    # find the 95% confidence interval
    $ci = $z * $std_err;

    $$min_ci_ref[$s] = $mean - $ci;
    $$max_ci_ref[$s] = $mean + $ci;

}

###############################################################################
```

```
# BuildDataLine
#
# Desc:
#   Calculate the value of a specified datapoint.
#
#   Create a line for a data file that has multiple indices
#   with one series per index.  This is a common datafile format for
#   all graph types
#
# Input:
#   $pdata_ref
#   $gconfig_ref
#   $s            data series
#   $x            current x value
#   $y            current y value
#   $verbose
#
# Output:
#   $data_ref  data file line
#
##################################################################################
sub BuildDataLine
{
    my($pdata_ref, $gconfig_ref, $s, $x, $y, $data_ref, $verbose)=@_;
    my($c);
    my($a);
    my($type);
    my($param);
    my($axis_label);
    my(@indep_labels);
    my(@min_data);
    my(@max_data);
    my(@min_ci);
    my(@max_ci);
    my(@avg_data);
    my($cur_data);
    my($min_indep_axes) = 1;
    my($max_indep_axes) = 2;
    my($dependent_label);
    my($head) = "# ";
    my($data);

    $max_indep_axes = 2 if ($$gconfig_ref{surface});
    $min_indep_axes = 2 if ($$gconfig_ref{constx});
    $max_indep_axes = 1 if ($$gconfig_ref{consty});

    # write independent series as first column(s) and subsequent
    # columns are the {data} array reference from the pdata structure
    foreach $type (sort keys %{$pdata_ref->[$x][$y][0]})
    {
        next if ($type =~ /pass/);

        # fill independent axes field
        AXES: for ($a = $min_indep_axes; $a <= $max_indep_axes; $a++)
        {
            foreach $param(sort keys %{$pdata_ref->[$x][$y][0]{$type}})
            {
                next if ($param =~ /raw/);
                next if ($param =~ /data/);

                # find independent variable for x-axis
                if ($pdata_ref->[$x][$y][0]{$type}{$param}[3] == $a)
                {
                    # handle independent axis labels
                    if (&IsNextIndex($data_ref, $verbose))
                    {
```

```
                    &PrettyAxisLabel($param, \$axis_label, $verbose);

                    $head .= sprintf("%s ", $axis_label);

                    # add labels to beginning of array
                    $indep_labels[$a - $min_indep_axes] = $axis_label;
                }

                # use current value of the parameter that matched
                $data .= sprintf("%d ",
                scalar($pdata_ref->[$x][$y][0]{$type}{$param}[2]));

                # save the initial and step value for axis tic increment
                $pdata_ref->[0][0][0]{text}{tics}[$a] =
                    [ (@{$pdata_ref->[$x][$y][0]{$type}{$param}}) ];

                next AXES;
            }
        }
    }
}

# insert the independent labels at beginning of label array
# and print a heading
if (! $$data_ref)
{
    $pdata_ref->[0][0][0]{text}{labels} =
        [ (@indep_labels, @{$pdata_ref->[0][0][0]{text}{labels}}) ];
    $$data_ref = "# @{$pdata_ref->[0][0][0]{text}{labels}}\n\n\n";
}

# if no experiment was run, do not build a data line
if (! defined $pdata_ref->[$x][$y][0]{text}{data})
{
    # chomp the previous newline since we will add another one
    chomp($$data_ref);
    return;
}

@avg_data = ();
# Run through all cycles for the current execution pass
# to build an array of sums that will be used to calculate
# the average
#
# That is, loop through all elements in the data line
for ($c = 0; $c < @{$pdata_ref->[$x][$y]}; $c++)
{
    $cur_data = $pdata_ref->[$x][$y][$c]{text}{data}[$s];

    # calculate an array of minimum and maximum values for errorbars
    $min_data[$s] = $cur_data
        if (($cur_data < $min_data[$s]) || ($min_data[$s] == 0));
    $max_data[$s] = $cur_data if ($cur_data > $max_data[$s]);

    # build an array of sums
    $avg_data[$s] += $cur_data;
}


# print the actual benchmark data
    # calculate the average by dividing each summation result
    # by the total number of cycles
$avg_data[$s] /= @{$pdata_ref->[$x][$y]};


# find the confidence interval
```

```
    if ($$gconfig_ref{confbars} == 1)
    {
        &ConfidenceInterval($pdata_ref, $avg_data[$s], $s, $x, $y,
            \@min_ci, \@max_ci, $verbose);
    }

    # label each index at beginning of index block
    if (&IsNextIndex($data_ref, $verbose))
    {
        $dependent_label = $s + ($max_indep_axes - $min_indep_axes) + 1;
        # the label array includes the independent axis labels
        $$data_ref .= $head . sprintf("%s min max min_ci max_ci (series %d) ",
            $pdata_ref->[0][0][0]{text}{labels}[$dependent_label], $s)."\n";
    }

    $$data_ref .= $data . sprintf("%.1f %.1f %.1f %.1f %.1f",
        $avg_data[$s], $min_data[$s], $max_data[$s],
        $min_ci[$s], $max_ci[$s]);

}


###############################################################################
# GenerateDatFile
#
# Desc:
# Generate gnuplot dat file with automatically detected independent data series
#
# Input:
#    $pdata_ref
#    $dat_file
#    $verbose
#
# Output:
#
###############################################################################
sub GenerateDatFile
{
    my($pdata_ref, $gconfig_ref, $dat_file, $verbose) = @_;
    my($i);
    my($x);
    my($y);
    my($xinit);
    my($xmax);
    my($yinit);
    my($ymax);
    my($data) = "";

    if ($$gconfig_ref{surface})
    {
        $xinit = 0; $xmax = @{$pdata_ref} - 1;
        $yinit = 0; $ymax = @{$pdata_ref->[$x]} - 1;
    }
    else
    {
        if ($$gconfig_ref{constx})
        {
            $xinit = $xmax = $$gconfig_ref{constx};
            $yinit = 0; $ymax = @{$pdata_ref->[$x]} - 1;
        }
        elsif ($$gconfig_ref{consty})
        {
            $yinit = $ymax = $$gconfig_ref{consty};
            $xinit = 0; $xmax = @{$pdata_ref} - 1;
        }
    }
    # to support errorbars, split multiseries data into multiple
    # indices with associated error columns
    for ($i = 0; $i < @{$pdata_ref->[0][1][0]{text}{data}}; $i++)
    {
        # for 2D plots, the x loop may execute only once
        for ($x = $xinit; $x <= $xmax; $x++)
        {
            # for 2D plots, the y loop may execute only once
            for ($y = $yinit; $y <= $ymax; $y++)
            {
                &BuildDataLine($pdata_ref, $gconfig_ref, $i, $x, $y,
                    \$data, $verbose);

                $data .= "\n";          # normal newline for each record
            }

            if ($$gconfig_ref{surface} == 1)
            {
                $data .= "\n";          # single blank line between datablocks
            }
        }

        if ($$gconfig_ref{surface} == 0)
        {
            $data .= "\n";
        }
        $data .= "\n";                  # double blank line between indices
    }

    printf "DATA:\n";
    printf "$data";

    open(FH, ">>$dat_file")
        || &ReportError("$0: Cannot open $dat_file\n", $verbose);
    printf FH "$data";
    close(FH);

}


################################################################################
# SetTics
#
# Desc:
# Calculate the tics initial position and increment
#
# Input:
#    $pdata_ref
#    $gconfig_ref
#    $verbose
#
# Output:
#
################################################################################
sub SetTics
{
    my($pdata_ref, $gconfig_ref, $verbose) = @_;
    my(@tics_limit);               # lowest limit of tics for each axis
    my($i);
    my($min_indep_axes) = 1;
    my($max_indep_axes) = 2;
    my($initial);                  # may be upper or lower increment
    my($increment);
    $max_indep_axes = 2 if ( $$gconfig_ref{surface });
    $min_indep_axes = 2 if ( $$gconfig_ref{constx });
    $max_indep_axes = 1 if ( $$gconfig_ref{consty });

    ($tics_limit[1]) = ( $$gconfig_ref{xrange} =~ /(\d+):/);
    ($tics_limit[2]) = ( $$gconfig_ref{yrange} =~ /(\d+):/);

    # $i represents the x and possibly y independent axes
    for ( $i = $min_indep_axes; $i <= $max_indep_axes; $i++)
    {
        $initial = $pdata_ref->[0][0][0]{ text}{ tics }[$i][0];
        $increment = $pdata_ref->[0][0][0]{ text}{ tics }[$i][1];
        $initial =~ s/[A-Za-z]//g;      # remove any units
        $increment =~ s/[A-Za-z]//g;    # remove any units

        if ( $increment < 0)     # find lower initial tic if negative increment
        {
            $increment = -$increment;
            while ( $initial >= $tics_limit[$i])
            {
                $initial -= $increment;
            }
            $initial += $increment;
        }

        $pdata_ref->[0][0][0]{ text}{ tics }[$i] = [ ( $initial, $increment) ];
    }

}


################################################################################
# GuessZLabel
#
# Desc:
# Heuristic that uses supplied graph ranges to position z-axis label
# This feature is not supported by gnuplot's zlabel command
#
# Input:
#    $pdata_ref
#    $gconfig_ref
#    $zlabel_coord_ref
#    $verbose
#
# Output:
#
################################################################################
sub GuessZLabel
{
    my($pdata_ref, $gconfig_ref, $zlabel_coord_ref, $verbose) = @_;

    my( @xlimits );
    my( @ylimits );
    my( @zlimits );
    my($xspan );
    my($yspan );
    my($zspan );

    @xlimits = split(/:/, $$gconfig_ref{xrange });
    @ylimits = split(/:/, $$gconfig_ref{yrange });
    @zlimits = split(/:/, $$gconfig_ref{zrange });

    $xspan = $xlimits[1] - $xlimits[0];
    $yspan = $ylimits[1] - $ylimits[0];
    $zspan = $zlimits[1] - $zlimits[0];
    # best guess factors to place rotated z-axis label
#   $xspan *= 1.15;                   # for reverse view
#   $yspan *= 0.1;                    # for reverse view
    $xspan *= 0.1;                   # for standard view
    $yspan *= 1.7 if ($yspan < 320); # for standard view
    $yspan *= 1.5 if ($yspan >= 320);
    $yspan *= 1.05 if ($zspan > 100000); # wide z-tic labels
    $zspan = $zlimits[0] + $zspan * 0.3;

    $$zlabel_coord_ref = "$xspan,$yspan,$zspan";
        print "coords:  $$zlabel_coord_ref\n";
        print "units:  $pdata_ref->[0][0][0]{text}{units}[12]\n";

}

################################################################################
# CreateGnuplotHeader
#
# Desc:
# Construct a string for generic gnuplot settings
#
# Input:
#    $pdata_ref
#    $dat_file
#    $gconfig_ref
#    $verbose
#
# Output:
#    $plot_cmd
#
################################################################################
sub CreateGnuplotHeader
{
    my($pdata_ref, $gconfig_ref, $dat_file, $verbose) = @_;
    my($plot_cmd);
    my($eps_file);
    my($xlabel_index) = 0;
    my($ylabel_index) = 1;
    my($zlabel_index) = 2;
    my($zlabel_coord);
    my($min_indep_axes) = 1;

    ($eps_file = $dat_file) =~ s/\.dat$/\.eps/;

    # set the initial and increment value of the tics
    &SetTics($pdata_ref, $gconfig_ref, $verbose);

    if ($$gconfig_ref{surface} == 0)
    {
        # use series field to find appropriate index and add offset of 1
        # for the single independent axis in the label array
        $ylabel_index = $$gconfig_ref{series} + 1;
    }
    else
    {
        $zlabel_index = $$gconfig_ref{series} + 2;
    }

    # build a generic header for constant settings
    # 2003/10/9 added global font at end of this set statement
    # $plot_cmd .= "set terminal postscript eps color solid \"Times-Roman\" 18\n";
    $plot_cmd .= "set terminal postscript eps color solid \"Helvetica\" 13\n";
    #$plot_cmd .= "set terminal x11\n";
    #$plot_cmd .= "set logscale y 10\n";
    #$plot_cmd .= "\"Times-Roman,24\"\n";
    $plot_cmd .= "set output \"$eps_file\"\n";
    $plot_cmd .= "set xrange [$$gconfig_ref{xrange}]\n";
    $plot_cmd .= "set yrange [$$gconfig_ref{yrange}]\n";
    $plot_cmd .= "set zrange [$$gconfig_ref{zrange}]\n";

    # needed to locate index of tics if two dimensional plot and const x
    $min_indep_axes = 2 if ($$gconfig_ref{constx});

    # set the axes tics
    $plot_cmd .= "set xtics mirror " .
        "$pdata_ref->[0][0][0]{text}{tics}[$min_indep_axes][0], " .
        "$pdata_ref->[0][0][0]{text}{tics}[$min_indep_axes][1]\n";
    if ($$gconfig_ref{surface} == 1)
    {
        $plot_cmd .= "set ytics mirror $pdata_ref->[0][0][0]{text}{tics}[2][0], ".
                     "$pdata_ref->[0][0][0]{text}{tics}[2][1]\n";
        $plot_cmd .= "set ztics mirror\n";
        $plot_cmd .= "set ticslevel 0.05\n";
        $plot_cmd .= "set nokey\n";
        $plot_cmd .= "set format y \"%g     \"\n";
        $plot_cmd .= "set format x \"  %g\"\n";
    }
    else
    {
        $plot_cmd .= "set format x \"%g\"\n";
        $plot_cmd .= "set border 31 lw 1.2\n"; # all four borders
    }

    # set the axes labels
    $plot_cmd .= "set xlabel \"" .
                 $pdata_ref->[0][0][0]{text}{labels}[$xlabel_index] .
                 "\" \"Times-Roman,18\"\n";

    if ($$gconfig_ref{surface} == 0)
    {
        if ($$gconfig_ref{label})
        {
            # manually overriding automatic labels due to multiple series
            $plot_cmd .= "set ylabel \"" . $$gconfig_ref{label} .
                 "\" \"Times-Roman,18\"\n";
        }
        else
        {
            $plot_cmd .= "set ylabel \"" .
                 $pdata_ref->[0][0][0]{text}{labels}[$ylabel_index] . " (" .
                 $pdata_ref->[0][0][0]{text}{units}[$ylabel_index -1] . ")" .
                 "\" \"Times-Roman,18\"\n";
        }
    }
    elsif ($$gconfig_ref{surface} == 1)
    {
        &GuessZLabel($pdata_ref, $gconfig_ref, \$zlabel_coord, $verbose);

        $plot_cmd .= "set ylabel \"" .
                 $pdata_ref->[0][0][0]{text}{labels}[$ylabel_index] .
                 "     \" \"Times-Roman,18\"\n";
        $plot_cmd .= "set label \"" .
                 $pdata_ref->[0][0][0]{text}{labels}[$zlabel_index] . " (" .
                 $pdata_ref->[0][0][0]{text}{units}[$zlabel_index -2] . ")" .
                 "\" at $zlabel_coord rotate center font \"Times-Roman,18\"\n";
    }

    return $plot_cmd;
}
```

```
################################################################################
# GeneratePlot
#
# Desc:
# Run gnuplot
#
# Input:
#     $pdata_ref
#     $gconfig_ref
#     $dat_file
#     $verbose
#
# Output:
#
################################################################################
sub GeneratePlot
{
    my($pdata_ref, $gconfig_ref, $dat_file, $verbose) = @_;
    my($plot_cmd);
    my($index);
    my($series);
    my($title);
    my($linetype);
    my(@titles);
    my(@series_list);

    $plot_cmd = &CreateGnuplotHeader($pdata_ref, $gconfig_ref,
        $dat_file, $verbose);

    $title = "Bonnie";

    if ($$gconfig_ref{surface} == 1)
    {
        # custom settings for 3D surface plot
        # rotate around initial x-axis (x & y in screen plane)
        $plot_cmd .= "set surface\n";
        $plot_cmd .= "set grid x y z\n";
#       $plot_cmd .= "set view 60, 240\n";     # original orientation
        $plot_cmd .= "set view 60, 330\n";     # standard view
#       $plot_cmd .= "set view 60, 150\n";     # reverse view
        $plot_cmd .= "set hidden3d\n";
        $plot_cmd .= "set border 127+256+512\n";
        $plot_cmd .= "set pm3d\n";
        $plot_cmd .= "set palette\n";
        $plot_cmd .= "unset colorbox\n";

        # build plot command
        $plot_cmd .= "splot ";
        $plot_cmd .= "'$dat_file' ";
        $plot_cmd .= "index $$gconfig_ref{series} using 1:2:3 ";
        $plot_cmd .= "title \"$title\" with lines linetype 1 linewidth 4, ";
    }
    else
    {
        # XXX took out key for Joly
        $plot_cmd .= "set key left top Left reverse samplen 5 spacing 1.5\n";
        @titles = split(/,/, $$gconfig_ref{family});
        @series_list = split(/,/, $$gconfig_ref{series});

        $plot_cmd .= "plot ";

        for ($i = 0; $i < @titles; $i++)
        {
            if ($$gconfig_ref{label})
            {
                # multiple series within one data set
            $series = $series_list[$i];
        }
        else
        {
            # multiple data sets, but same series in each
            $series = @{$pdata_ref->[0][0][0]{text}{data}} * $i +
                $$gconfig_ref{series};
        }

        $linetype = $i + 1;

        $plot_cmd .= "'$dat_file' ";
        $plot_cmd .= "index $series using 1:2 ";
        $plot_cmd .= "title \"$titles[$i]\" ";
        $plot_cmd .= "with lines linetype $linetype linewidth 4, ";
        # if errorbars are requested, a second plot command is needed
        if ($$gconfig_ref{errorbars} == 1)
        {
            $plot_cmd .= "'$dat_file' ";
            $plot_cmd .= "index $series using 1:2:3:4 ";
            $plot_cmd .= "notitle ";
            $plot_cmd .= "with yerrorbars linetype $linetype linewidth 4, ";
        }
        # errorbars may be based on 95% confidence interval instead
        elsif ($$gconfig_ref{confbars} == 1)
        {
            $plot_cmd .= "'$dat_file' ";
            $plot_cmd .= "index $series using 1:2:5:6 ";
            $plot_cmd .= "notitle ";
            $plot_cmd .= "with yerrorbars linetype $linetype linewidth 4, ";
        }
    }
  }

  $plot_cmd =~ s/\,$/\n/;        # trim off extra comma

  print "$plot_cmd\n";
  open GP, "|$gnuplot -persist" || die("oops\n");
  print GP "$plot_cmd";
  close GP;
}

################################################################################
# dispono
#
# Desc:
#     type parameter initial value   step value
# Configure system for ESDC experiments
#
# Input:
#     $raw_file
#     $dat_file
#     $gconfig_ref
#     $verbose
#
# Output:
#     @users
#
################################################################################
sub Dispono
{
  my($raw_file, $dat_file, $gconfig_ref, $verbose) = @_;
  my(@pdata);
  my($bench_cmd);
  my($cycles);
```

```
    my( @raw_files );
    my( $file );

    if ( $raw_file =~ / ,/)
    {
        @raw_files = split (/ ,/ , $raw_file );
    }
    else
    {
        $raw_files [0] = $raw_file ;
    }

    if (-e "$dat_file")
    {
        unlink("$dat_file");
    }

    foreach $file ( @raw_files)
    {
        @pdata = ();

        # Load raw benchmark output file and populate pdata structure
        &ParseRawFile (\ @pdata , \ $bench_cmd , $file , $verbose );

        # Determine benchmark type and extract data from raw benchmark text
        &ParseBenchmarkText(\ @pdata , \ $bench_cmd , $dat_file , $verbose );

        # Build dat file for gnuplot
        &GenerateDatFile (\ @pdata , $gconfig_ref , $dat_file , $verbose );
    }

    # Create a line plot , errorbar plot , or surface plot
    &GeneratePlot (\ @pdata , $gconfig_ref , $dat_file , $verbose );

}

################################################################################
# GraphConfig
#
# Desc :
#    General graph configuration options
#
# Input :
#    $gconfig_ref
#    $series
#    $multi
#    $errorbars
#    $confbars
#    $verbose
#
# Output :
#    @users
#
################################################################################
sub GraphConfig
{
    my( $gconfig_ref , $series , $bisect , $xrange , $yrange , $zrange , $family ,
        $label , $errorbars , $confbars , $verbose ) = @_;
    my( $constx , $consty );

    ( $constx , $consty ) = [illegible]lit (/ ,/ , $bisect );

    $$gconfig_ref{series } = $series ;         # data series or index number
    $$gconfig_ref{errorbars } = $errorbars ; # enable errorbars (min-max)
    $$gconfig_ref{confbars } = $confbars ;   # enable errorbars (C.I.)
    $$gconfig_ref{family } = $family ;       # titles for family of curves
    $$gconfig_ref{label} = $label;          # manual y-axis label for multi-series
    $$gconfig_ref{constx} = $constx;        # constant x-axis for y-z plane
    $$gconfig_ref{consty} = $consty;        # constant y-axis for x-z plane
    $$gconfig_ref{xrange} = $xrange;        # plotting range for x-axis
    $$gconfig_ref{yrange} = $yrange;        # plotting range for y-axis
    $$gconfig_ref{zrange} = $zrange;        # plotting range for z-axis
    $$gconfig_ref{surface} = 1              # a three dimensional plot
        if (($constx == 0) && ($consty == 0));

}


################################################################################
# ReportError
#
# Desc:
#    Print error message.
#
# Input:
#    $message    - error message
#    $display    - flag to control display of error message
#
# Output:
#    none
#
################################################################################
sub ReportError
{
    my($message, $display) = @_;

    print STDERR $message if ($display);
    exit(1);
} # ReportError


################################################################################
# Usage
#
# Desc:
#    Print usage statement.
#
# Input:
#    $ret_code    - program exit code
#
# Output:
#    exits with $ret_code
#
################################################################################
sub Usage
{
    my($ret_code) = @_;

    print "\nUsage: $0 -r <raw_file> -d <dat_file> -s <series> -b <bisect>";
    print " [-x <xrange>] [-y <yrange>] [-z <zrange>] [-f <family_titles>]";
    print " [-e] [-c] [-q] [-h]\n";
    print "\nwhere\n";
    print "    -r raw benchmark data file\n";
    print "    -d dat file created for plotting\n";
    print "    -s select data series for plotting\n";
    print "    -b 2D plot for const x (y-z plane), const y (x-z plane) \n";
    print "    -x colon-separated xrange of plot \n";
    print "    -y colon-separated yrange of plot \n";
    print "    -z colon-separated zrange of plot \n";
    print "    -f comma-separated list of series titles \n";
    print "    -l manual y-axis label if multiple series\n";
    print "    -e enable errorbars showing minimum and maximum data\n";
    print "    -c enable errorbars for 95% confidence interval\n";
    print "    -q quiet\n";
    print "    -h this help\n";
    print "    \n";
    print "    NOTE: - default plot style is a surface plot\n";
    print "            where <constx> and <consty> are zero (b=0,0)\n";
    print "    \n";
    print "          - multiple raw files (comma sep) needed for family\n";
    print "    \n";
    print "    \n";

    exit $ret_code;
} # Usage
```